Prof. Dr. H. Seidl

Skript zur Vorlesung

Analyse von Programmen

Dieses Skript entstand während der Vorlesung *Analyse von Programmen*, die im Sommersemester 1995 an der Universität Trier gelesen wurde. LaTeX–Version by Frank Burch.

Inhaltsverzeichnis

Kapitel 1

Einführung

Die Aufgabe der Programmanalyse ist es, Laufzeiteigenschaften von Programmen zu bestimmen, ohne jedoch das Programm auszuführen.
Die dadurch gewonnenen Informationen erweisen sich als nützlich

1. im Compilerbau, um besseren Code zu erzeugen. D.h. z.B.
   - überflüssige Berechnungen zu vermeiden;
   - Spezialfälle zu erkennen, in denen das Codeerzeugungsschema verbessert werden kann;
   - bessere Einrichtungen der Hardware (Register, Caches, Pipelines, Multiprozessor–Systeme) besser auszunutzen;
2. in Programmentwicklungsumgebungen, um Hinweise auf Programmierfehler zu finden;
3. beim Reverse–Engineering (z.B. zur Analyse schlecht dokumentierter Programme).

Wir interessieren uns in dieser Darstellung nur für Punkt 1.

Folgende Probleme treten in diesem Zusammenhang auf:

Korrektheit: Um die Korrektheit der Analyse zu gewährleisten, ist eine Formalisierung der Semantik von Programmen erforderlich.

Präzision: Alle Fragen, die wir stellen, sind unentscheidbar. Wir müssen uns deshalb mit einer 'sicheren' Approximation begnügen.

Effizienz: Eine effiziente Analyse erfordert gute Algorithmen und 'raffinierte' Tricks zur geschickten Implementierung.

Kapitel 2

Analyse imperativer Programme

Interessante Fragen sind hier:

- Welche Werte haben die Variablen an Programmpunkten ?

  Konstantenpropagation: Ist der Wert einer Variablen stets gleich (z.B. x=5) ? In diesem Fall braucht die Variable nicht angelegt zu werden. Die Berechnung des R–Werts wird ersetzt durch das Laden einer entsprechenden Konstante (hier 5).

  Intervallanalyse: Ändert sich der Variablenwert nur innerhalb eines bestimmten Intervalls (z.B. bei Feldern)? Für Indexausdrücke kann man so feststellen, ob zwei Iterationen über das gleiche Feld unabhängig sind.

- Wie verhält sich das Programm ?

  Dead–code–elimination: Werden bestimmte Programmteile je betreten ?

  Lifeness: Wird eine Variable ab einem bestimmten Programmpunkt noch gebraucht ? Wenn nicht, kann der Variablenplatz freigegeben, sowie Zuweisungen an die Variable eliminiert werden.

  Definedness: Wurde eine bestimmte Variable bereits definiert ?

  Available expressions: Wurde der Wert eines Teilausdrucks *exp* bereits berechnet ? Wenn ja, so muß dieser Ausdruck nur einmal gespeichert und kein weiteres Mal berechnet werden.

<u>Die Grammatik für Programme:</u>

p	::=	`begin` block `end`	
block	::=	decl; block | stats	
decl	::=	`var` x: τ	
stats	::=	state | state; stats	
state	::=	`read`(x) | `write`(exp) | `nop` | $x := exp$	(elementare Statements)
state	::=	`if` b `then` stats `else` stats `fi` | `while` b `do` stats `od`	(zus.gesetzte Statements)

wobei x der Name einer Variablen, τ der Typ `integer` oder `boolean` und b, exp Ausdrücke sind. Wir verwenden keine Pointer, Felder oder Prozeduren.

Beispiel: Berechnung der Fakultät

$p \equiv$ `begin`
  `var` x, y `:integer;`
  `read` (x);
  $y := 1$;
  `while` $(x > 0)$ `do`
    $y := y * x$;
    $x := x - 1$
  `od;`
  `write`(y)
  `end`

2.1 Operationelle Semantik von Programmen und der Kontrollflußgraph

Sei D die Menge der Werte (Integers und Booleans), $\perp \notin D$ bezeichne den 'undefinierten Wert'. Das Tupel $(\sigma, stats)$ heißt *Zustand einer Berechnung* und enthält für jede Variable ihren gegenwärtigen Wert, sowie die Folgen der Werte auf Eingabe I und Ausgabe O und einen Programmpunkt. Genauer:
Die *Variablenbelegung* σ bezeichnet eine Abbildung von deklarierten Variablen auf Werte in $D \cup \{\perp\}$, sowie von I bzw. O auf D^*. Die Menge aller dieser Abbildungen heiße Z.
Der *Programmpunkt stats* entspricht dem Teil des Programms, der noch abgearbeitet werden mu. Er wird im Folgenden durch eine Folge von Statements repräsentiert.
Vor der Berechnung von p enthält der Zustand eine Abbildung $\sigma_w,\ w \in D^*$

$$\begin{aligned} \sigma_w : x &\mapsto \perp, \qquad \text{für alle Variablen } x \\ I &\mapsto w \\ O &\mapsto \varepsilon \end{aligned}$$

Für eine Variablenbelegung $\sigma \in Z$ lät sich der Wert eines Ausdrucks $exp \in Expressions$ folgendermaen durch die Funktion $[.] : Expressions \longrightarrow (Z \longrightarrow D \cup \{\perp\})$ bestimmen:

$[\mathbf{c}](\sigma) = c$, $\mathbf{c}$ Konstante und $c \in D$
$[x](\sigma) = \sigma(x)$, x Variablenname
$[exp_1 \square exp_2](\sigma) = [exp_1](\sigma) \square [exp_2](\sigma),\ exp_1, exp_2 \in Expressions$

Beispiele:
$[1](\sigma) = 1$
$[1+2](\sigma) = 3$
$[x+y](\sigma) = [x](\sigma) + [y](\sigma) = \sigma(x) + \sigma(y) = 3 + 4 = 7$, falls $\sigma(x) = 3$ und $\sigma(y) = 4$
$[x+1](\sigma) = \perp$, falls $\sigma(x) = \perp$

Operatoren, angewandt auf $\perp$, liefern $\perp$.

Die Programmausführung besteht in einer Folge von Reduktionsschritten $(\sigma, s) \to (\sigma', s')$ wobei die Relation '$\to$' definiert ist durch

(1) s beginnt mit einem elementaren Statement s_0
$(\sigma, s_0; s') \to (\sigma', s')$ wobei σ' wie folgt definiert ist:

$s_0 \equiv \texttt{nop}: \ \sigma' = \sigma$

$$s_0 \equiv \texttt{read}(x): \ \sigma' \text{ ist die Funktion mit } \sigma'(y) = \begin{cases} w' & \text{für } y \equiv I \text{ (wobei } \sigma(I) = dw') \\ d & \text{für } y \equiv x \\ \sigma(y) & \text{sonst} \end{cases}$$

$$s_0 \equiv \texttt{write}(exp): \ \sigma' \text{ ist die Funktion mit } \sigma'(y) = \begin{cases} dw & \text{für } y \equiv O \quad \text{(wobei } \sigma(O) = w \\ \sigma(y) & \text{sonst} \quad\quad\quad \text{mit } d = [exp](\sigma)) \end{cases}$$

$$s_0 \equiv x := exp: \ \sigma' \text{ ist die Funktion mit } \sigma'(y) = \begin{cases} [exp](\sigma) & \text{für } y \equiv x \\ \sigma(y) & \text{sonst} \end{cases}$$

(2) s beginnt mit dem **if**–Statement, d.h. $s \equiv \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}; s'$, dann gilt:
$(\sigma, s) \to (\sigma, s_1; s')$ falls $[b](\sigma) = tt$ bzw.
$(\sigma, s) \to (\sigma, s_2; s')$ falls $[b](\sigma) = f\!f$.

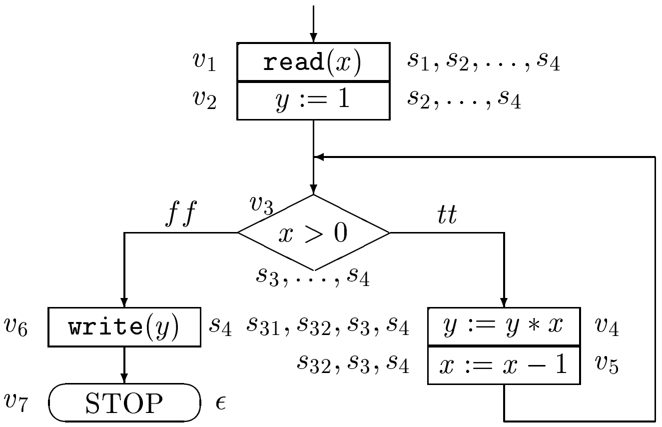
(3) s beginnt mit dem **while**–Statement, d.h. $s \equiv \texttt{while } b \texttt{ do } s_1 \texttt{ od}; s'$, dann gilt:
$(\sigma, s) \to (\sigma, s_1; s')$ falls $[b](\sigma) = tt$ bzw.
$(\sigma, s) \to (\sigma, s')$ falls $[b](\sigma) = f\!f$.

Die Berechnung endet, falls ein Zustand (σ, ϵ) erreicht wird.
($tt \widehat{=} true$ und $f\!f \widehat{=} false$).

Verzichten wir in der Relation '$\to$' auf die erste Komponente, so erhalten wir den *Kontrollfluß-graphen* G_p zum Programm p.

Beispiel:

```
p ≡ begin
      var x, y :integer;
      read (x);            (s1)
      y := 1;              (s2)
      while (x > 0) do     (s3)
        y := y * x;        (s31)
        x := x - 1         (s32)
      od;
      write(y)             (s4)
      end
```



Beachte:

- G_p läßt sich auch direkt aus dem Programmtext gewinnen (dies entspricht einer graphischen Repräsentation von p).
- Berechnungen des Programms p sind Wege im Graphen G_p. Terminierende Berechnungen sind Wege, die am Knoten ϵ enden.

Der Kontrollflußgraph G_p für das Programm p ist ein gerichteter Graph (V, E), dessen Knotenmenge die Menge der Programmpunkte von p ist; eine Kante $(s, s') \in E$ liegt vor, genau dann wenn $(\sigma, s) \to (\sigma', s')$ gilt für Variablenbelegungen σ, σ'. Zusätzlich definieren wir eine Markierung λ wie folgt:

$\lambda(s; s') = s$, falls s elementar
$\lambda(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}; s') = b$
$\lambda(\texttt{while } b \texttt{ do } s \texttt{ od}; s') = b$
$\lambda(\epsilon) = \text{STOP}$

2.2 Lebendigkeit von Variablen und Verfügbarkeit von Ausdrücken

Ein Knoten $v \in V$ heißt *Benutzung (use)* einer Variablen x, falls $\lambda(v)$ auf den Wert von x 'zugreift', d.h. x kommt im Ausdruck $\lambda(v)$ vor.
Ein Knoten $v \in V$ heißt *Definition* einer Variablen x, falls $\lambda(v)$ entweder die Form $\texttt{read}(x)$ hat oder eine Zuweisung $x := exp$ ist.
Eine Variable x heißt *lebendig relativ zu einem Pfad* π, der vom Knoten v nach ϵ führt ($\pi = v \cdots \epsilon$), falls es ein Präfix π' von π gibt, das von v zu einer Benutzung von x führt, aber selbst keine Definition von x enthält.
$LIFE[\pi]$ sei die Menge der relativ zu π lebendigen Variablen.

Beispiel:
$\pi = v_3\, v_4\, v_5\, v_3\, v_6\, v_7$ (siehe Beispiel Kontrollflußgraph)
$LIFE[\pi] = \{x, y\}$
(die Auswertung der rechten Seite erfolgt vor Zuweisung zur linken Seite)

Eine Variable x heißt *lebendig an einem Knoten* v, falls x lebendig ist relativ zu irgendeinem Pfad von v nach ϵ. $LIFE[v]$ bezeichne die Menge dieser Variablen. Offenbar gilt:

$$LIFE[v] = \bigcup_{\pi = v \cdots \epsilon} LIFE[\pi]$$

Berechnung von $LIFE[\pi]$:
Seien $D_v = \mathcal{P}(\{x \mid x \text{ ist Programmvariable}\})$ und $\text{Var}(exp)$ die Menge aller Variablen, die im Ausdruck exp vorkommen. Die Berechnung von $LIFE[\pi]$ erfolgt induktiv über die Pfadlänge von π, beginnend im Knoten ϵ mit $LIFE[\epsilon] = \emptyset$.
Idee: Ordne nun jedem $\rho \in Bild(\lambda)$ eine Funktion $[\rho] : D_v \longrightarrow D_v$ zu, die die Lebendigkeit relativ zum Pfad $v\pi'$ mit $\lambda(v) = \rho$ aus der Lebendigkeit relativ zu einem Pfad π' berechnet, sodaß $LIFE[v\,\pi'] = [\rho](LIFE[\pi'])$ gilt.
$[\texttt{nop}] = id$ (d.h. $[\texttt{nop}](A) = A$)
$[exp](A) = \text{Var}(exp) \cup A$

$[\mathtt{write}(exp)](A) = \mathrm{Var}(exp) \cup A$
$[\mathtt{read}(x)](A) = A \setminus \{x\}$
$[x := exp](A) = (\mathrm{Var}(exp) \cup A) \setminus \{x\}$

Beachte: Die Berechnung von $LIFE[\pi]$ erfolgt 'vom Programmende' aus.

Verfügbare Ausdrücke:

Ein Knoten v heißt *Berechnung* eines Ausdrucks exp, falls der Wert von exp an v berechnet wird. D.h.:

$\lambda(v) \equiv exp$ oder
$\lambda(v) \equiv \mathtt{write}(exp)$ oder
$\lambda(v) \equiv y := exp$

Ein Ausdruck exp heißt *verfügbar (available) relativ zu einem Pfad* π von dessen Anfang zum Knoten v, falls es ein Suffix π' von π gibt, das von einer Berechnung von exp zu v führt und keine Definition einer Variablen, die in exp vorkommt, enthält. Wir bezeichnen die Menge aller Ausdrücke, die relativ zu einem Pfad π verfügbar sind, als $AV[\pi]$.

Beispiel:

$\pi = v_1\, v_2\, v_3\, v_4\, v_5 \quad \Rightarrow \quad AV[\pi] = \{'1'\}$

Berechnung von $AV[\pi]$:

Seien die Mengen $D_e = \mathcal{P}(Expressions)$ und $E_x = \{exp \in Expressions \mid exp$ enthält die Variable $x\}$ gegeben. Die Funktion $[\rho] : D_e \longrightarrow D_e$ ist dann wie folgt definiert ($\rho \in Bild(\lambda)$):

$[\mathtt{nop}] = id$
$[b](A) = A \cup \{b\}$
$[\mathtt{write}(exp)](A) = A \cup \{exp\}$
$[\mathtt{read}(x)](A) = A \setminus E_x$
$[x := exp](A) = (A \cup \{exp\}) \setminus E_x$
$[\mathrm{STOP}] = id$

Wir berechnen $AV[\pi]$ wie folgt ($\pi = \epsilon$ ist der leere Pfad):

$$AV[\epsilon] = \emptyset \text{ und } AV[\pi' v] = [\lambda(v)](AV[\pi'])$$

Die Menge $AV[v]$ der nach einem Knoten v verfügbaren Ausdrücke ist dann definiert durch:

$$AV[v] = \bigcap_{\substack{\pi \text{ ist Pfad vom} \\ \text{Startknoten nach } v}} AV[\pi]$$

Ein Ausdruck ist also am Knoten v verfügbar, falls er relativ zu **jedem** Pfad vom Startknoten nach v verfügbar ist.
Hinweis: Um auch über verfügbare Ausdrücke **vor** dem ersten Knoten des Kontrollflußgraphen sprechen zu können, wird dieser i. a. um einen explizit anzugebenden Startknoten s erweitert (etwa mit Markierung `nop`).
$AV[v]$ wird in Abhängigkeit zu **allen** Pfaden vom Startknoten zu v definiert (Vorwärtsproblem).

Zur Konstruktion effektiver Verfahren benötigen wir einige Grundbegriffe aus der Theorie partiell geordneter Mengen.

2.3 Partielle Ordnungen und Verbände

Definition: Sei D eine Menge und $\sqsubseteq\ \subseteq D \times D$ eine zweistellige Relation über D.
Eine *partielle Ordnung (partial ordered set (po))* ist ein Paar $(D, \sqsubseteq)$ für das gilt:

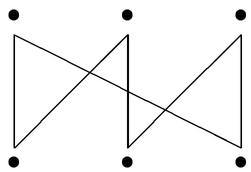
1. $x \sqsubseteq x \quad \forall x \in D$ (Reflexivität)
2. $x \sqsubseteq y$ und $y \sqsubseteq x \Rightarrow x = y \quad \forall x, y \in D$ (Antisymmetrie)
3. $x \sqsubseteq y$ und $y \sqsubseteq z \Rightarrow x \sqsubseteq z \quad \forall x, y, z \in D$ (Transitivität)

Bemerkung: Ist '$\sqsubseteq$' bekannt, dann schreiben wir auch D statt $(D, \sqsubseteq)$.

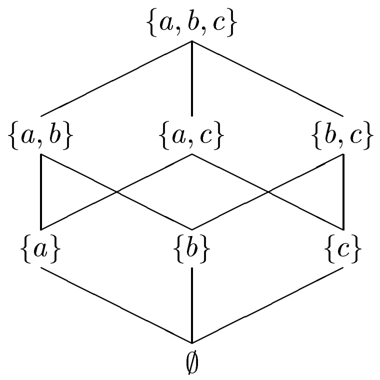
Beispiel für partielle Ordnungen: (HASSE–Diagramme)

0 1 2 3 · · ·

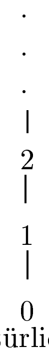
1) 'flache Ordnung' auf $(\mathbb{N}, =)$



2) 3–Crown



3) Teilmengenverband



4) $\mathbb{N}$ mit 'natürlicher Ordnung'

Unter $\mathbb{N}$ verstehen wir hier immer die natürlichen Zahlen einschließlich der 0.
Die strukturerhaltenden Abbildungen zu partiellen Ordnungen sind die monotonen Funktionen.

Definition: Seien $(D_1, \sqsubseteq)$ und $(D_2, \sqsubseteq)$ partielle Ordnungen und f eine Funktion $f : D_1 \longrightarrow D_2$.
Dann heißt f *monoton*, falls für alle $x, y \in D_1$ gilt: $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$

Die Algorithmen, die wir konstruieren werden, erzeugen i. a. Folgen $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots$ von 'Approximationen' an den zu berechnenden Wert.

Definition: Sei $(D, \sqsubseteq)$ eine partielle Ordung, $X \subseteq D$.
$d \in D$ heißt *obere Schranke* von X, falls $x \sqsubseteq d \ \ \forall x \in X$. Eine obere Schranke d heißt *kleinste obere Schranke (least upper bound (lub))* von X, falls $d \sqsubseteq d'$ für jede obere Schranke d' von X.

Wir schreiben für kleinste obere Schranken: $d = \bigsqcup X$ oder $d = x \sqcup y$, falls $X = \{x, y\}$ oder $d = \bigsqcup\limits_{i \geq 0} x_i$, falls $X = \{x_i \mid i \geq 0\}$.

Analog definiert man untere Schranken, größte untere Schranken (greatest lower bounds (glb)) und notiert diese mit $\sqcap$.

Definition: Eine *(aufsteigende) Kette (ascending chain (ac))* in einer partiellen Ordnung $(D, \sqsubseteq)$ ist eine (endliche oder unendliche) Folge von Elementen $x_0, x_1, x_2, \ldots \in D$ mit $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$.

Definition: Eine partielle Ordnung $(D, \sqsubseteq)$ heißt *vollständige partielle Ordnung (complete partial order (cpo))*, falls sie die beiden folgenden Eigenschaften besitzt:

1. Es gibt ein kleinstes Element $\bot = \bigsqcap D \in D$
2. Jede aufsteigende Kette $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ besitzt eine kleinste obere Schranke in D.

In den Beispielen zu partiellen Ordnungen sind 3),5),6),7),8) und 9) vollständig.

Die strukturerhaltenden Abbildungen zu vollständigen partiellen Ordnungen sind die stetigen Funktionen.

Definition: Seien $(D_1, \sqsubseteq)$ und $(D_2, \sqsubseteq)$ vollst. partielle Ordnungen und f eine Funktion $f : D_1 \longrightarrow D_2$.
f heißt *stetig*, falls für jede Kette $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ aus D_1 gilt: $f(\bigsqcup_{i \geq 0} x_i) = \bigsqcup_{i \geq 0} f(x_i)$

Bemerkung: Aus der Stetigkeit einer Funktion f folgt stets deren Monotonie.

Beispiel:

$$\text{Die Funktion } succ : I\!N_\bot \longrightarrow I\!N_\bot \text{ mit } succ(x) = \begin{cases} \bot, & \text{falls } x = \bot \\ x+1, & \text{sonst} \end{cases} \quad \text{ist stetig.}$$

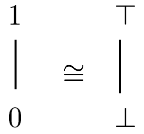
Definition: Eine partielle Ordnung $(D, \sqsubseteq)$ heißt *vollständiger Verband (complete lattice (cl))*, falls zu jeder (endlichen oder unendlichen) Teilmenge $X \subseteq D$ eine kleinste obere Schranke $\bigsqcup X \in D$ existiert.

Satz: In einem vollständigen Verband $(D, \sqsubseteq)$ hat jede Menge $X \subseteq D$ eine größte untere Schranke $\bigsqcap X$.

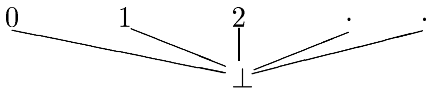
Beweis:
Sei $X \subseteq D$ und $U_X = \{d \in D \mid d \sqsubseteq x \ \ \forall x \in X\}$ die Menge aller unteren Schranken von X.
Offenbar gilt $U_X \neq \emptyset$, da $\bot := \bigsqcup \emptyset \in U_X$.
Betrachte $d = \bigsqcup U_X$. Für d gilt:

1. d ist untere Schranke von X, denn:
   Sei $x \in X$ beliebig. Dann gilt: $y \sqsubseteq x \ \ \forall y \in U_X \implies x$ ist obere Schranke von $U_X \implies d \sqsubseteq x$, da d kleinste obere Schranke von U_X
2. d ist größte untere Schranke von X, denn:
   Sei d' beliebige untere Schranke von X, dann gilt: $d' \in U_X \implies d' \sqsubseteq \bigsqcup U_X = d$ □

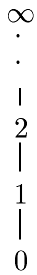
<u>Weitere Beispiele:</u>



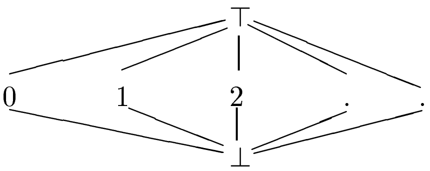
5) **2**–Ordnung



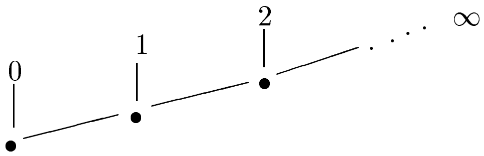
6) 'flache' cpo $I\!N_\perp$



7) $\mathcal{N}$



8) 'flacher' Verband $I\!N_\perp^\top$



9) Lazy–$I\!N$

Die Beispiele 3),5),7) und 8) sind vollständige Verbände.

<u>Satz:</u> (Knaster–Tarski) Sei $(D, \sqsubseteq)$ ein vollständiger Verband.
Dann besitzt jede monotone Funktion $f : D \longrightarrow D$ einen kleinsten Fixpunkt d. D.h. $f(d) = d$ und $\forall d' \in D$ mit $f(d') = d'$ gilt $d \sqsubseteq d'$.

<u>Beweis:</u>
Sei $F = \{x \in D \mid f(x) \sqsubseteq x\}$ und $d := \bigsqcap F$ $(*)$.
Es gilt $d \in F$, denn $d \sqsubseteq x \quad \forall x \in F \stackrel{f \text{ mon.}}{\Longrightarrow} f(d) \sqsubseteq f(x) \sqsubseteq x \quad \forall x \in F \Longrightarrow f(d)$ ist untere Schranke von $f \stackrel{(*)}{\Longrightarrow} f(d) \sqsubseteq d \Longrightarrow d \in F$.

1. $f(d) = d$, denn: Annahme: $f(d) \sqsubset d$ (d.h. $f(d) \sqsubseteq d$ und $f(d) \neq d$) $\stackrel{f \text{ mon.}}{\Longrightarrow} f(f(d)) \sqsubseteq f(d) \Longrightarrow f(d) \in F \Longrightarrow d$ ist keine untere Schranke von $F \Longrightarrow$ Widerspruch $\Longrightarrow d$ ist Fixpunkt von f
2. d ist kleinster Fixpunkt von f, denn:
   Sei d' ein weiterer Fixpunkt von $f \Longrightarrow f(d') \sqsubseteq d' \Longrightarrow d' \in F \Longrightarrow d \sqsubseteq d'$ □

<u>Satz:</u> (Kleene) Sei $(D, \sqsubseteq)$ eine vollständige partielle Ordnung. Dann besitzt jede stetige Funktion $f : D \longrightarrow D$ einen kleinsten Fixpunkt $d \in D$.

<u>Beweis:</u> Definiere eine Folge d_i, $i \geq 0$ durch $d_0 = \perp$ und $d_i = f(d_{i-1})$ für $i > 0$.

1. d_i, $i \geq 0$ ist aufsteigende Kette in D, denn:
   $d_0 = \perp \sqsubseteq d_1$

Für $i > 0$ nehmen wir an, daß $d_{i-1} \sqsubseteq d_i$ gilt, dann ist $d_i = f(d_{i-1}) \sqsubseteq f(d_i) = d_{i+1}$ (da f monoton).

2. $d := \bigsqcup_{i \geq 0} d_i$ ist kleinster Fixpunkt von f, denn:
   - $f(d) = f(\bigsqcup_{i \geq 0} d_i) = \bigsqcup_{i \geq 0} f(d_i) = \bigsqcup_{i \geq 0} d_{i+1} = \perp \sqcup \bigsqcup_{i > 0} d_i = \bigsqcup_{i \geq 0} d_i = d$
   - d ist kleinster Fixpunkt von f, d.h. $d \sqsubseteq x \ \ \forall x \in D$ mit $f(x) = x$:
     Sei $x \in D$ mit $f(x) = x$. Zeige: $\bigsqcup_{i \geq 0} d_i \sqsubseteq x$, d.h. $d_i \sqsubseteq x \ \ \forall i \geq 0$:
     $i = 0 : \perp \sqsubseteq x$
     $i > 0 :$ Annahme $d_i \sqsubseteq x \overset{f \text{ mon.}}{\Longrightarrow} d_{i+1} = f(d_i) \sqsubseteq f(x) = x$ □

Dieser Satz gestattet die Berechnung des kleinsten Fixpunktes einer Funktion f nur dann , falls die Folge $\perp, f(\perp), f^2(\perp), \ldots$ irgendwann stationär wird ($d := \bigsqcup_{i \geq 0} f^i(\perp)$).
Dies ist insbesondere dann der Fall, wenn D die Ascending–Chain–Condition erfüllt.

Definition: Eine partielle Ordnung D erfüllt die *Ascending–Chain–Condition (acc)*, falls für jede aufsteigende Kette $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots$ in D ein j existiert mit $d_j = d_{j'} \ \ \forall j' \geq j$.

Beispiel:
$I\!N_\perp, I\!N_\perp^\top$ und **2** erfüllen die acc, $\mathcal{N} = \{0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq \cdots \sqsubseteq \infty\}$ und Lazy–$I\!N$ erfüllen die acc nicht.

Konstruktion neuer po's, cpo's und cl's aus bereits bestehenden:
Definition: Es seien D_1 und D_2 partielle Ordnungen.
Dann ist $D_1 \times D_2$ die partielle Ordnung mit $(d_1, d_2) \sqsubseteq (d_1', d_2') \iff d_i \sqsubseteq d_i', \ \ i = 1, 2$.
Im Weiteren sind $[D_1 \to D_2]_m$ die *Menge aller monotonen* und $[D_1 \to D_2]_c$ die *Menge aller stetigen Funktionen*, die von D_1 nach D_2 abbilden. Diese Mengen werden angeordnet durch die sogenannte *punktweise Anordnung*: $f_1 \sqsubseteq f_2 \iff f_1(x) \sqsubseteq f_2(x) \ \ \forall x \in D_1$. $[D_1 \to D_2]_c$ wird auch mit $[D_1 \to D_2]$ bezeichnet.

Satz: Seien D_1 und D_2 partielle Ordnungen. Dann gilt:

1. Sind D_1 und D_2 vollständige partielle Ordnungen (vollständige Verbände), dann ist auch $D_1 \times D_2$ eine vollständige partielle Ordnung (ein vollständiger Verband).
2. Erfüllen D_1 und D_2 die Ascending–Chain–Condition, dann erfüllt auch $D_1 \times D_2$ diese.

Beweis: (von 2.)
Sei $(d_0, e_0) \sqsubseteq (d_1, e_1) \sqsubseteq (d_2, e_2) \sqsubseteq \cdots$ eine aufsteigende Kette in $D_1 \times D_2$.
Dann sind $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots$ und $e_0 \sqsubseteq e_1 \sqsubseteq e_2 \sqsubseteq \cdots$ aufsteigende Ketten in D_1 bzw. D_2.
Da D_1 und D_2 die Ascending–Chain–Condition erfüllen, gilt:
$\exists j_1$ so daß $d_j = d_{j_1} \ \ \forall j \geq j_1$ und $\exists j_2$ so daß $d_j = d_{j_2} \ \ \forall j \geq j_2$
Wähle nun $j' := j_1 \sqcup j_2$, dann ist $(d_j, e_j) = (d_{j'}, e_{j'}) \ \ \forall j \geq j' \Longrightarrow$ Behauptung □

Satz: Seien D_1 und D_2 partielle Ordnungen. Dann gilt:

1. Ist D_2 eine vollständige partielle Ordnung, dann ist auch $[D_1 \to D_2]_m$ eine vollständige partielle Ordnung.
2. Sind D_1 und D_2 vollständige partielle Ordnungen, dann ist auch $[D_1 \to D_2]_c$ eine vollständige partielle Ordnung.

Wir geben nun Methoden an, um auf einfache Weise stetige Funktionen zu gewinnen.

Konstruktion stetiger Abbildungen:

1. **Komposition**: Sind $f_1 : D_1 \longrightarrow D_2$ und $f_2 : D_2 \longrightarrow D_3$ stetige Funktionen, dann ist auch $f_2 \circ f_1 : D_1 \longrightarrow D_3$ eine stetige Funktion.

2. **Kartesisches Produkt**: Die Abbildung $f : D_1 \times D_2 \longrightarrow D$ ist genau dann stetig, wenn die Funktionen $f_x : D_2 \longrightarrow D,\ \ x \in D_1$ und $f^y : D_1 \longrightarrow D,\ \ y \in D_2$, stetig sind.
Dabei ist $f_x(d_2) = f(x, d_2)$ und $f^y(d_1) = f(d_1, y)$.
Insbesondere ist für stetige $f_i : D_i \longrightarrow D_i',\ \ i = 1, 2$, die Funktion $f_1 \times f_2 : D_1 \times D_2 \longrightarrow D_1' \times D_2'$ stetig, wobei $(f_1 \times f_2)(d_1, d_2) = (f_1(d_1), f_2(d_2))$.

3. **Bausteine zur Konstruktion stetiger Funktionen**: Seien im Folgenden $D, D_1, \ldots, D_n$ vollständige partielle Ordnungen.

   - $id : D \longrightarrow D$ mit $id(d) = d$ ist stetig (Identität);
   - $\pi_i : D_1 \times \cdots \times D_n \longrightarrow D_i,\ n \geq 1,\ 1 \leq i \leq n$ mit $\pi_i(d_1, \ldots, d_n) = d_i$ ist stetig (Projektion);
   - $\Delta_n : D \longrightarrow D^n$ mit $\Delta(d) = (d, \ldots, d),\ d \in D$ ist stetig (Diagonalfunktion);
   - $const_c : D_1 \longrightarrow D_2,\ \ c \in D_2$, mit $const_c(d) = c,\ \ \forall d \in D_1$ ist stetig (Konstante).
   - Seien Ω eine Menge von stetigen Operatoren $\Box : D^k \longrightarrow D$ (i. a. ist $k \leq 2$) und $Vars$ eine Menge von Variablen. Dann repräsentiert jeder Ausdruck exp, aufgebaut aus Konstanten aus D und Variablen aus $Vars$ mit Hilfe von Operatoranwendungen von Elementen aus Ω, eine stetige Funktion: $[exp] : [Vars \rightarrow D]_m \longrightarrow D$ (als partielle Ordnung auf $Vars$ nimmt man die flache Ordnung $(=)$)
     Beweis: (Induktion über den Aufbau eines Ausdrucks)
     $[\mathbf{c}] = const_c$ ist stetig, wobei $\mathbf{c}$ Konstante, $c \in D$
     $[x] = \pi_x$ ist stetig, wobei $x \in Vars$
     $[exp_1 \Box exp_2]$ ist stetig, falls $[exp_1], [exp_2]$ stetig.
     Es gilt nämlich $[exp_1 \Box exp_2] = \Box \circ ([exp_1] \times [exp_2]) \circ \Delta_2$, denn:

$$
\begin{aligned}
(\Box \circ ([exp_1] \times [exp_2]) \circ \Delta_2)(\eta) &= (\Box \circ ([exp_1] \times [exp_2])(\eta, \eta) \\
&= \Box([exp_1](\eta), [exp_2](\eta)) \\
&= [exp_1](\eta) \Box [exp_2](\eta) \\
&= [exp_1 \Box exp_2](\eta)
\end{aligned}
$$

Beispiel:
Sei M Menge, $D := \mathcal{P}(M)$ die Potenzmenge von M und

$$\cup, \cap : D^2 \longrightarrow D$$

$$(.) \backslash A : D \longrightarrow D,\quad A \subseteq D$$

$$;: D^2 \longrightarrow D \text{ definiert durch } A_1; A_2 = \begin{cases} \emptyset & \text{, falls } A_1 = \emptyset \\ A_2 & \text{, sonst} \end{cases}$$

seien stetige Operatoren. Dann ist jede Funktion, die wir als Ausdruck mit Hilfe dieser Operatoren konstruieren können, stetig.

Anwendung:
Lösen von Gleichungssystemen S mit $x = f_x, \ x \in Vars$,
wobei f_x Ausdrücke über $Vars$, Konstanten und stetigen Operatoren aus Ω sind.
Es gilt folgender Satz:

Satz: Ein Gleichungssystem S besitzt eine kleinste Lösung $\eta : Vars \longrightarrow D$ wobei D eine cpo ist. η ist gegeben durch $\eta = \bigsqcup_{j \geq 0} \eta_j$, wobei $\eta_0(x) = \bot, \ \forall x \in Vars$ und $\eta_j(x) = [f_x](\eta_{j-1})$ für $j > 0$ und $x \in Vars$.

Beweis: D ist cpo $\implies$ $(Vars \to D)$ ist ebenfalls cpo.
Da Ω nur stetige Operatoren enthält, ist $[f_x]$ stetig für alle $x \in Vars$
$\implies$ $F : (Vars \to D) \longrightarrow (Vars \to D)$, definiert durch $F(\eta')(x) := [f_x](\eta')$, ist ebenfalls stetig
$\implies$ F besitzt einen kleinsten Fixpunkt; dieser ist gleich η
$\implies$ η ist kleinste Lösung von S $\square$

Kosten der Berechnung von η:
Annahmen:

- Operatoranwendungen der Form $d_1 \Box d_2$, sowie Vergleiche (z. B. $(d_1 == d_2)$), können in konstanter Zeit (also in O(1)) ausgeführt werden.
- Setze $|S| = \sum_{x \in Vars} (1 + |f_x|)$, wobei $|f_x|$ die Größe des Ausdrucks f_x ist (also der Anzahl der Zeichen in f_x entspricht).
- D erfülle die acc, genauer: jede **echt** aufsteigende Kette $d_0 \sqsubset d_1 \sqsubset d_2 \sqsubset \cdots$ in D sei höchstens von der Länge h (unter der Länge einer Kette verstehen wir die Anzahl ihrer Elemente).

Unter diesen Voraussetzungen schließen wir:

1. Die Länge der längsten echt aufsteigenden Kette in $(Vars \to D)$ ist $\sharp Vars \cdot (h-1) + 1$ (der Beweis sei dem interessierten Leser als Übung empfohlen).
2. Jede Iteration kostet $O(|S|)$

Damit erhält man für die Laufzeit: $O(h \cdot \sharp Vars \cdot |S|)$

Beispiel: Sei folgendes Gleichungssystem mit $D = \mathcal{P}(\{a, b, c\})$ gegeben:

$$\begin{aligned} x_1 &= x_2 \cap x_4 \\ x_2 &= (x_2 \cup \{a, b\}) \cap x_3 \\ x_3 &= x_4 \cup \{a, c\} \\ x_4 &= (x_3 \cap \{a, b\}) \cup \{c\} \end{aligned}$$

Var	Anzahl der Iterationen				
	0	1	2	3	4
x_1	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	$\{a\}$
x_2	$\emptyset$	$\emptyset$	$\{a\}$	$\{a\}$	$\{a\}$
x_3	$\emptyset$	$\{a, c\}$	$\{a, c\}$	$\{a, c\}$	$\{a, c\}$
x_4	$\emptyset$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	$\{a, c\}$

Man benötigt hier 16 Neuberechnungen der Variablen.

Eine erste Verbesserung erreicht man folgendermaßen:

- Ordne die Menge der Variablen an (z. B. DFS–Ordnung), d. h. $Vars = \{x_1, \ldots, x_n\}$!
- Verwende bei der Berechnung von x_j bereits die **neuen** Werte für x_i, $i < j$ (ordne die Variablen dazu nach ihren Abhängigkeiten voneinander) !

Beispiel: Bei Verwendung der Anordnung $x_4 \to x_3 \to x_2 \to x_1$ erhält man:

Var	Anzahl der Iterationen			
	0	1	2	3
x_4	$\emptyset$	$\{c\}$	$\{a,c\}$	$\{a,c\}$
x_3	$\emptyset$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$
x_2	$\emptyset$	$\{a\}$	$\{a\}$	$\{a\}$
x_1	$\emptyset$	$\emptyset$	$\{a\}$	$\{a\}$

Man benötigt hier lediglich 12 Neuberechnungen der Variablen.

Algorithmus:

```
for (i := 0; i < n; i++)    /* Vars = {x_0, x_1, ..., x_{n-1}} */
  η(x_i) := ⊥;   /* Initialisierung von η */
repeat {
  finished := 1;
  for (i := 0; i < n; i++) {
    t := [f_{x_i}](η);
    if (t != η(x_i)) {
      η(x_i) := t;
      finished := 0;
    }
  }
} until (finished);
```

Komplexität des Algorithmus':

Sei η_j der Wert von η nach dem j–ten Durchlauf der `repeat`–Schleife. dann gilt:

1. Für jedes i ist $\eta_0(x_i) \sqsubseteq \eta_1(x_i) \sqsubseteq \cdots$ (Beweis mittels Induktion über die Anzahl der Iterationen unter Verwendung der Monotonie von $[f_{x_i}]$).
2. Falls die `repeat`–Schleife **nicht** nach dem j–ten Durchlauf endet, war $finished == 0$, d. h. $\eta_j(x_i) \neq \eta_{j-1}(x_i)$ für ein i.
   $\Longrightarrow$ die `repeat`–Schleife wird maximal $n \cdot (h-1)$–mal durchlaufen $\Longrightarrow$ die Laufzeit ergibt sich zu $O(n \cdot h \cdot |S|)$

Nun stellt sich die Frage, ob wir hier überhaupt etwas gewonnen haben.
Da unsere Abschätzung sehr grob war, müssen wir eine genauere Betrachtung vornehmen:

- Ist S nicht rekursiv, so benötigt die `repeat`–Schleife (bei DFS–Anordnung) nur eine Iteration !
- Enthält S nur **eine** 'Rückwärtsabhängigkeit', werden höchstens h Iterationen benötigt.

Allgemeiner: Sei m die Anzahl der f_{x_i}, die Variablen x_j enthalten mit $j \geq i$. Dann ist die Anzahl der Iterationen $\leq m \cdot (h-1)+1$, d. h. die Laufzeit ist $O((m \cdot h+1) \cdot |S|)$ (wobei in realen Anwendungen $m \ll n$ ist).

Korrektheit des Algorithmus':

Sei $\bar{\eta}$ die kleinste Lösung des Gleichungssystems S. Sei N die Anzahl der Iterationen der `repeat`–Schleife. Dann gilt:

1. $\eta_j \sqsubseteq \bar{\eta} \;\; \forall j \geq 0$ (Induktion nach j)
2. η_N ist Lösung von S, da $finished == 1$ ist bei Verlassen der `repeat`–Schleife $\implies \eta_N = \bar{\eta}$

Beobachtung: Bei der Berechnung der nächsten Iteration werden oft gleiche Werte immer wieder berechnet. Eine weitere Verbesserung erreichen wir nun, indem wir nur die Änderungen propagieren.

Der Worklist–Algorithmus:

```
forall (x ∈ Vars)    /* Initialisierung von η */
  η(x) := ⊥;
W := Vars;   /* Initialisierung der Worklist */
while (W  != ∅) {
  x :=extract(W);    /* erstes Element aus Worklist entfernen */
  t := [f_x](η);
  if (t  != η(x)) {
    η(x) := t;    /* Update η(x) */
    W := W∪̇{y | f_y enthält x};
  }
}
```

W kann beispielsweise als Schlange implementiert werden. Die Funktion **extract** liefert das erste Element der Schlange. Das Einfügen ($\dot{\cup}$) von Elementen erfolgt nur dann, wenn diese noch nicht in der Schlange enthalten sind.

Beispiel:
Wir verwenden wieder die Variablenanordnung $x_4 \rightarrow x_3 \rightarrow x_2 \rightarrow x_1$.

W	Neuberechnungen
x_4, x_3, x_2, x_1	$x_4 = \{c\}$
x_3, x_2, x_1	$x_3 = \{a, c\}$
x_2, x_1, x_4	$x_2 = \{a\}$
x_1, x_4, x_2	$x_1 = \emptyset$
x_4, x_2	$x_4 = \{a, c\}$
x_2, x_3, x_1	$x_2 = \{a\}$
x_3, x_1	$x_3 = \{a, c\}$
x_1	$x_1 = \{a\}$
$\emptyset$	–

Man benötigt hier lediglich 8 Neuberechnungen der Variablen.

Komplexität des Worklist–Algorithmus':

Die Folge der Werte $\eta(x)$ ist aufsteigend. $\Longrightarrow$ Die Bedingung des `if`–Statements kann für eine Variable x höchstens $(h-1)$–mal erfüllt sein.
Sei $n_x := k$ das k–malige Vorkommen von x in den rechten Seiten des Gleichungssystems S.
Dann ist $\sum\limits_{x\in Vars} (h-1)\cdot n_x$ eine obere Schranke für die Anzahl der Variableneinfügungen in der Worklist W. Folglich ist die Anzahl der Extraktionen aus W:

$$\sharp Vars + \sum_{x\in Vars} (h-1)\cdot n_x \leq |S| + (h-1)\cdot \sum_{x\in Vars} n_x \leq |S| + (h-1)\cdot |S| = h\cdot |S|$$

$\Longrightarrow$ Die Laufzeit ist von der Größenordnung:

$$\mathrm{O}\underbrace{(h\cdot|S|)}_{\substack{\text{Variablen}-\\\text{einfügungen}}} + \mathrm{O}\underbrace{(h\cdot|S|)}_{\text{Extraktionen}} + \mathrm{O}\underbrace{(h\cdot e\cdot|S|)}_{\substack{\text{Berechnung}\\\text{von }[f_x](\eta)}} + \mathrm{O}\underbrace{(h\cdot|S|)}_{\text{Rest}} = \mathrm{O}(h\cdot e\cdot|S|),$$

wobei e eine obere Schranke für die Zeit zur Auswertung einer rechten Seite von S ist (i. a. ist $e = \mathrm{O}(1)$).

Korrektheit des Worklist–Algorithmus':

Sei N die Anzahl der Schleifendurchläufe mit η_j als Variablenbelegung und W_j als Worklist nach j–tem Durchlauf. Dann gilt:

1. $\eta_j \sqsubseteq \bar{\eta} \;\; \forall j \geq 0 \quad$ ($\bar{\eta}$ ist kleinste Lösung von S)
2. $\{x \mid \eta_j(x) \neq [f_x](\eta)\} \subseteq W_j \;\; \forall j \geq 0$

Da $W_N = \emptyset$ ist, ist also η_N eine Lösung von S. □

Praktischer Vergleich der beiden letzten Verfahren:

- Verfahren 1 ist einfacher, hat aber (zumindest theoretisch) schlechtere Laufzeit
- Verfahren 2 ist etwas komplizierter, hat aber (zumindest theoretisch) bessere Laufzeit

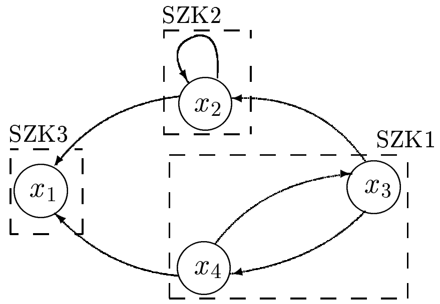
Vereinigung beider Verfahren:
Idee:

- Berechne zuerst die starken Zusammenhangskomponenten (SZK) des Abhängigkeitsgraphen für S !
- Behandle jeweils eine starke Zusammenhangskomponente gesondert mit dem Worklist–Algorithmus !
- Propagiere die Ergebnisse !
- Behandle die nächste starke Zusammenhangskomponente !

Laufzeit: $O(|S|+h\cdot e\cdot c)$, mit $c=\sum\limits_{x\in Vars}|f_x|$, wobei für diese Summe nur Variablen x betrachtet werden, die in nichtrivialen SZK liegen.

Beispiel: Abhängigkeitsgraph zu unserem Beispiel oben

Berechne der Reihe nach Lösungen für SZK x_3, x_4, SZK x_2 und SZK x_1.



W	Neuberechnungen	
x_3, x_4	$x_3=\{a,c\}$	
x_4	$x_4=\{a,c\}$	Lösung für SZK1
x_3	$x_3=\{a,c\}$	
x_2	$x_2=\{a\}$	Lösung für SZK2
x_2	$x_2=\{a\}$	
x_1	$x_1=\{a\}$	Lösung für SZK3

Man benötigt hier lediglich 6 Neuberechnungen der Variablen.

2.4 Anwendung: Approximation von 'Merge over all Paths'

Sei D ein vollständiger Verband und $\mathcal{F}\subseteq[D\to D]_c$ eine Menge sog. *Transferfunktionen*.
Sei $G=(V,E)$ ein gerichteter Graph ($V=\{s,v_0,v_1,\ldots,v_n\}$), $s\in V$ der Startknoten mit Eingangsgrad 0, von dem aus alle anderen Knoten des Graphen erreichbar sind.
Sei weiter $d\in D$ ein Startwert für s und die Funktion $\mu: V\backslash\{s\}\longrightarrow\mathcal{F}$ eine Zuordnung von Knoten auf Transferfunktionen an diesem Knoten. Dann gilt:
Wir können μ auf Pfaden fortsetzen durch $\mu(v_0\cdots v_k):=\mu(v_k)\circ\cdots\circ\mu(v_0)$ ($\mu(\varepsilon)=id$).
Für $v\in V$ sei $\Pi(v)$ die Menge der Pfade von s nach v ohne den Knoten s, wobei $\Pi(s)=\{\varepsilon\}$.
Wir möchten nun den MOP

$$X_v=\bigsqcup_{\pi\in\Pi(v)}\mu(\pi)(d),\ v\in V\qquad\text{berechnen.}$$

Für die *Berechnung verfügbarer Ausdrücke* gilt dann:
$D=(\mathcal{P}(Expressions),\supseteq)$ (die 'natürliche' Ordnung ist nun umgekehrt !).
$\mathcal{F}$ ist die Menge der Funktionen die beschrieben wird durch Abbildungen $f(x)=(A\cup x)\cap B$ mit $A,B\subseteq Expressions$ und $x\in D$.
G ist der Kontrollflußgraph KFG mit Startknoten s.
$AV[v]=:X_v$ ist die Menge der verfügbaren Ausdrücke nach Besuch von v.
Bei der Berechnung verfügbarer Ausdrücke definierten wir beispielsweise u. a. eine (stetige) Funktion $[x:=exp]: D\longrightarrow D$ mit $[x:=exp](A):=(A\cup\{exp\})\setminus E_x=(A\cup\{exp\})\cap E_x^c$, wobei $exp\in Expressions$, E_x Menge der Ausdrücke, welche die Variable x enthalten und E_x^c das Komplement von E_x bezüglich $Expressions$ sind. Für diese Funktion gilt offenbar $[x:=exp]\in\mathcal{F}$.

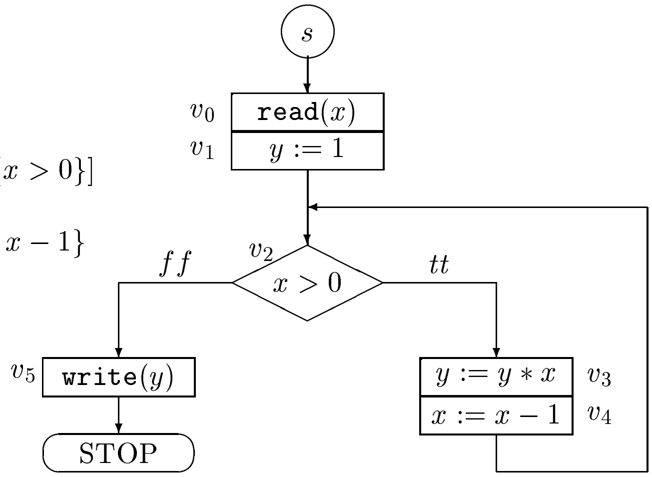
Für die *Berechnung lebendiger Variable* gilt dann:
$D=(\mathcal{P}(Vars),\subseteq)$
$\mathcal{F}=\{f(x)=(A\cap x)\cup B\mid A,B\subseteq Vars,\ x\in D\}$
$G=\text{KFG}^R$ (man erhält KFG^R, indem man alle Kanten von KFG umkehrt).
$LIFE[v]=:X_v$ ist die Menge der lebendigen Variablen vor dem Knoten v.

Idee: Löse das Gleichungssystem S mit

$$x_s = d \text{ und } x_v = \bigsqcup_{(u,v)\in E} \mu(v)(x_u),\ v \neq s.$$

Beispiel: verfügbare Ausdrücke für $Expressions = \{1, x > 0, x * y, x - 1, y\}$

$AV[s] = \emptyset$
$AV[v_0] = AV[s] \backslash \{x > 0, x * y, x - 1\}$
$AV[v_1] = (AV[v_0] \cup \{1\}) \backslash \{x * y, y\}$
$AV[v_2] = (AV[v_1] \cup \{x > 0\}) \cap (AV[v_4] \cup \{x > 0\})$
$AV[v_3] = (AV[v_2] \cup \{x * y\}) \backslash \{x * y, y\}$
$AV[v_4] = (AV[v_3] \cup \{x - 1\}) \backslash \{x > 0, x * y, x - 1\}$
$AV[v_5] = AV[v_2] \cup \{y\}$



Beim MOP betrachten wir **alle Pfade** vom Startknoten zu einem Knoten v. Bei diesem Gleichungssystem betrachten wir die **direkten Vorgängerknoten** von v, von denen aufgrund der Definition des Graphen G bekannt ist, daß zu jedem dieser Vorgängerknoten ein Pfad von s zu eben diesem Knoten existiert.
Das nun folgende Koinzidenztheorem macht nun Aussagen darüber, wie gut dieses Gleichungssystem den MOP approximiert.

Definition: Eine Funktion $f : D \longrightarrow D$ heißt *distributiv*, falls $f(d_1 \sqcup d_2) = f(d_1) \sqcup f(d_2)\ \ \forall d_1, d_2 \in D$ gilt.

Satz: (Koinzidenztheorem)
Sei η kleinste Lösung des Gleichungssystems S mit $x_s = d \in D,\ D$ vollständiger Verband und $x_v = \bigsqcup_{(u,v)\in E} \mu(v)(x_u),\ v \neq s,\ \mu(v) \in \mathcal{F}$. Dann gilt:

1. $\eta(x_v) \sqsupseteq X_v,\ \forall v \in V$
2. Sind die Funktionen aus $\mathcal{F}$ **distributiv**, dann gilt sogar Gleichheit.
   D. h.: $\eta(x_v) = X_v,\ \ \forall v \in V$

Beweis:

1. Mittels Induktion nach der Länge von $\pi = v_1 \cdots v_n$ ist zu zeigen: $\mu(\pi)(d) \sqsubseteq \eta(x_v)\ \ \forall \pi \in \Pi(v)$
   $\pi = \varepsilon$: $\bigsqcup_{\pi\in\Pi(s)} \mu(\pi)(d) = \mu(\varepsilon)(d) = id(d) = d = \eta(x_s)$
   Andernfalls ist $\pi = \pi' v,\ v \neq s$, wobei $\pi' \in \Pi(w),\ (w,v) \in E$.
   Wir nehmen an, daß $\mu(\pi')(d) \sqsubseteq \eta(x_w)$
   Dann ist $\mu(\pi)(d) = \mu(\pi' v)(d) = \mu(v)(\mu(\pi')(d)) \overset{\text{Ind.vor.}}{\sqsubseteq} \mu(v)(\eta(x_w)) \sqsubseteq \bigsqcup_{(u,v)\in E} \mu(v)(\eta(x_u)) = \eta(x_v)$

2. Es genügt zu zeigen: Die Abbildung $Vars \to D$ mit $x_v \mapsto X_v,\ v \in V$ ist Lösung des Gleichungssystems
   $v = s$: $X_s = d = \eta(x_s)$
   $v \neq s$:

$$\begin{aligned} X_v &= \bigsqcup_{\pi \in \Pi(v)} \mu(\pi)(d) = \bigsqcup_{(w,v)\in E} \; \bigsqcup_{\pi' \in \Pi(w)} \mu(v)(\mu(\pi')(d)) \\ &\overset{(*)}{=} \bigsqcup_{(w,v)\in E} \mu(v) \left(\bigsqcup_{\pi' \in \Pi(w)} \mu(\pi')(d) \right) = \bigsqcup_{(w,v)\in E} \mu(v)(X_w) \end{aligned}$$

((*) da $\mu(v)$ distributiv ist, und deshalb mit nichtleeren, abzählbaren lubs kommutiert (Beweis s. Übung))
$\Longrightarrow$ $x_v \mapsto X_v$ ist Lösung $\square$

Anmerkung:

- Beispielsweise sind die Funktionen aus $\mathcal{F}$ mit $f(x) = (A \cup x) \cap B$ distributiv:

$$\begin{aligned} f(x_1 \cup x_2) &= (A \cup x_1 \cup x_2) \cap B \\ &= (A \cup x_1 \cup A \cup x_2) \cap B \\ &= (A \cup x_1) \cap B \cup (A \cup x_2) \cap B \\ &= f(x_1) \cup f(x_2) \end{aligned}$$

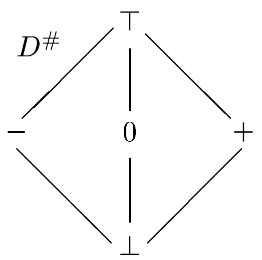
- Es gibt sinnvolle Analysen, die nichtdistributive Funktionen $\mu(v)$ benötigen.
- Auf jeden Fall sind wir aber beim Lösen des Gleichungssystems auf der sicheren Seite. Die Lösung des Gleichungssystems liefert evtl. gröbere Information als der MOP.

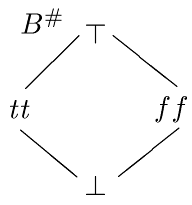
2.5 Datenabstraktion

Lebendigkeit von Variablen und Verfügbarkeit von Ausdrücken sind Eigenschaften der **Programmausführung** und unabhängig von den konkreten Werten der Variablen. Um Informationen über die konkreten Werte zur Ausführungszeit zu gewinnen, führen wir die Berechnungen des Programms nicht mit den 'konkreten Daten', sondern mit 'abstrakten Daten' aus.
Dies soll hier exemplarisch anhand der *Vorzeichenanalyse* und der *Konstantenpropagation* gezeigt werden.

2.5.1 Vorzeichenanalyse

Abstrakte Werte für Integers $D := \mathbb{Z}_\perp$ und Booleans B:





($\bot \,\hat{=}$ 'undefiniert' und $\top \,\hat{=}$ 'keine Aussage möglich')

Jedem Operator $\Box$ auf konkreten Werten ordnen wir einen abstrakten Operator $\Box^{\#}$ auf abstrakten Werten zu.

Beispiele:

$$\begin{array}{c|ccc} +^{\#} & - & 0 & + \\ \hline - & - & - & \top \\ 0 & - & 0 & + \\ + & \top & + & + \end{array} \qquad \begin{array}{c|ccc} *^{\#} & - & 0 & + \\ \hline - & + & 0 & - \\ 0 & 0 & 0 & 0 \\ + & - & 0 & + \end{array} \qquad \begin{array}{c|ccc} \leq^{\#} & - & 0 & + \\ \hline - & \top & tt & tt \\ 0 & ff & tt & tt \\ + & ff & ff & \top \end{array}$$

wobei $\bot \Box^{\#} d = d \Box^{\#} \bot = \bot$ und $\top \Box^{\#} d' = d' \Box^{\#} \top = \top$ für $\bot \neq d' \in D^{\#}$.

Abstrakte Variablenbelegungen sind Abbildungen $\sigma^{\#} : Vars \longrightarrow D^{\#} \cup B^{\#}$.
Analog zur 'normalen' Semantik, können wir eine abstrakte Semantik $[exp]^{\#}$ für einen Ausdruck $exp \in Expressions$ angeben:

$$[c]^{\#}(\sigma^{\#}) = \begin{cases} - & \text{falls } c < 0 \\ 0 & \text{falls } c = 0 \\ + & \text{falls } c > 0 \end{cases} \qquad c \in D$$

$[x]^{\#}(\sigma^{\#}) = \sigma^{\#}(x)$, x Variable

$[exp_1 \Box exp_2]^{\#}(\sigma^{\#}) = [exp_1]^{\#}(\sigma^{\#}) \Box^{\#} [exp_2]^{\#}(\sigma^{\#})$, $exp_1, exp_2 \in Expressions$

Fortsetzung auf Pfade π:
(s, v Knoten, SIGN$[\pi]$ abstrakte Variablenbelegung am Ende von Pfad π)

SIGN$[s]$, $x \to \bot$, $\forall x \in Vars$
SIGN$[\pi\ v] = [\lambda(v)]^{\#}(\text{SIGN}[\pi])$, wobei $[\text{STOP}]^{\#} = [\texttt{nop}]^{\#} = [\texttt{write}(exp)]^{\#} = [b]^{\#} = id$

$$[\texttt{read}(x)]^{\#}(\sigma^{\#}) = \sigma^{\#'} \text{ mit } \sigma^{\#'}(y) = \begin{cases} \top & \text{falls } y \equiv x \\ \sigma^{\#}(y) & \text{sonst} \end{cases}$$

$$[x := exp]^{\#}(\sigma^{\#}) = \sigma^{\#'} \text{ mit } \sigma^{\#'}(y) = \begin{cases} [exp]^{\#}(\sigma^{\#}) & \text{falls } y \equiv x \\ \sigma^{\#}(y) & \text{sonst} \end{cases}$$

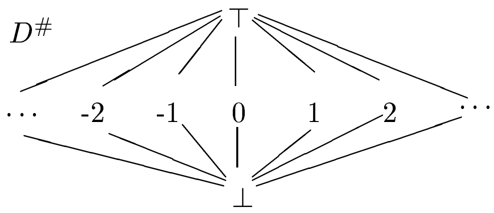
Schließlich ist $\text{SIGN}[v] := \bigsqcup_{\pi \in \Pi(v)} \text{SIGN}[\pi]$

Achtung: Die abstrakten Operationen sind nicht unbedingt distributiv !
Beispielsweise ist $+^{\#}(0,+) \cup +^{\#}(+,+) = (0 +^{\#} +) \sqcup (+ +^{\#} +) = + \sqcup + = +$ aber $+^{\#}((0,+) \sqcup (+,+)) = \top +^{\#} + = \top$, wobei $+^{\#} : D^{\#} \times D^{\#} \longrightarrow D^{\#}$

Gemäß der Aussage des Koinzidenztheorems liefert die Lösung des Gleichungssystems zu SIGN nicht unbedingt SIGN$[v]$, sondern eventuell ein größeres Ergebnis.

2.5.2 Konstantenpropagation

Wir gehen analog zur Vorzeichenanalyse vor. Als abstrakte Werte für Integers haben wir:



$D^{\#}$ ist zwar unendlich, erfüllt aber die acc.
Für die abstrakten Operatoren gilt:

$$x \,\Box^{\#} y := \begin{cases} x \,\Box\, y & \text{falls } x, y \in \mathbb{Z} \\ \bot & \text{falls } x = \bot \text{ oder } y = \bot \\ \top & \text{sonst} \end{cases}$$

Abstrakte Booleans sind analog zur Vorzeichenanalyse definiert.
Wir definieren auch $[exp]^{\#}$ analog für Ausdrücke. Allerdings ist nun $[c]^{\#}(\sigma^{\#}) = c,\ c \in D$
Die Definitionen von CONST$[\pi]$ und CONST$[v]$ sind völlig analog zu SIGN$[\pi]$ und SIGN$[v]$. Auch hier liefert ein entsprechendes Gleichungssystem eine obere Approximation an CONST$[v]$, $v \in V$. Leider sind die Operatoren wieder nicht unbedingt distributiv, wie das folgende Beispiel zeigt:

$$(2 +^{\#} 3) \sqcup (3 +^{\#} 2) = 5, \text{ aber } +^{\#}((2,3) \sqcup (3,2)) = +^{\#}(\top, \top) = \top +^{\#} \top = \top$$

Die kleinste Lösung des Gleichungssystems kann aber berechnet werden, da $D^{\#}$ und $B^{\#}$ die acc erfüllen.

2.5.3 Korrektheit von Vorzeichenanalyse und Konstantenpropagation

Wir führen eine Relation $\triangle \subseteq D \times D^{\#}$ zwischen konkreten und abstrakten Werten ein.
Für die Vorzeichenanalyse ist diese Relation:

$$a \triangle b^{\#} :\Longleftrightarrow \begin{cases} - \sqsubseteq b^{\#}, & \text{falls } a < 0 \\ 0 \sqsubseteq b^{\#}, & \text{falls } a = 0 \\ + \sqsubseteq b^{\#}, & \text{falls } a > 0 \\ tt & , \text{falls } a = \bot \end{cases} \qquad a \in D,\ b^{\#} \in D^{\#}$$

Für die Konstantenpropagation ist $\triangle$ definiert als:

$$a \triangle b^{\#} :\Longleftrightarrow\ a \sqsubseteq b^{\#} \quad (a \in D,\ b^{\#} \in D^{\#},\ \sqsubseteq \text{ Relation in } D^{\#})$$

Dabei soll für abstrakte Operatoren gelten:

$$a_1 \triangle b_1^{\#} \text{ und } a_2 \triangle b_2^{\#} \implies (a_1 \Box a_2) \triangle (b_1^{\#} \Box^{\#} b_2^{\#}),\ \forall a_1, a_2 \in D,\ b_1^{\#}, b_2^{\#} \in D^{\#}$$

D. h. $\Box \to \Box^{\#}$ ist *verträglich* mit $\triangle$.

In diesem Fall schreiben wir $\Box \triangle \Box^{\#}$ und nennen $\Box^{\#}$ *Abstraktion* von $\Box$ (relativ zu $\triangle$).
Diese Eigenschaft muß für jedes Operatorenpaar $(\Box, \Box^{\#})$ im Einzelfall nachgeprüft werden !

Gilt diese Eigenschaft, so müssen wir nachweisen, daß $[exp]\triangle[exp]^{\#}$ $\forall exp \in Expressions$ gilt.
D. h., falls $\sigma\triangle\sigma^{\#}$ erfüllt ist, dann soll gelten: $[exp](\sigma)\triangle[exp]^{\#}(\sigma^{\#})$
(wobei $\sigma\triangle\sigma^{\#} :\Longleftrightarrow \sigma(x)\triangle\sigma^{\#}(x) \;\; \forall x \in Vars$)
<u>Beweisskizze:</u>
Dazu ist zu zeigen:

1. $const_c\triangle const_{c^{\#}}$, falls $c\triangle c^{\#}$
2. Für $\pi_i : D_1 \times D_2 \longrightarrow D_i$ und $\pi_i^{\#} : D_1^{\#} \times D_2^{\#} \longrightarrow D_i^{\#}$ definiert durch $\pi_i(d_1, d_2) = d_i$, $\pi_i^{\#}(d_1^{\#}, d_2^{\#}) = d_i^{\#}$, $i = 1, 2$ gilt: $\pi_i\triangle\pi_i^{\#}$, wobei $(d_1, d_2)\triangle(d_1^{\#}, d_2^{\#}) \iff d_i\triangle d_i^{\#}$ für $i = 1, 2$.
3. $f\triangle f^{\#}$ und $g\triangle g^{\#} \implies (g \circ f)\triangle(g^{\#} \circ f^{\#})$ bzw. $(g \times f)\triangle(g^{\#} \times f^{\#})$

Aus diesen Eigenschaften folgern wir (analog zur Stetigkeit) die Behauptung. $\square$

Mittels Induktion nach der Länge eines Pfads $\pi = s\, v_1 \cdots v_n$ erhält man:
$(\sigma_0, s) \to (\sigma_1, v1) \to \cdots \to (\sigma_n, v_n)$ mit $\sigma_0 = \sigma_w \implies \sigma_n(x)\triangle P[\pi](x)$ mit $P \in \{\text{SIGN,CONST}\}$
(zur Erinnerung: $\sigma_w(I) = w,\ \sigma_w(x) = \bot$ ist die Anfangsbelegung von Variablen und Eingabe)
<u>Beweisskizze:</u>
Für $n = 0$ ist diese Behauptung richtig.
Für den Schritt $n-1 \to n$ soll $(\sigma, v) \to (\sigma', v')$ mit $\sigma(x)\triangle\sigma^{\#}(x),\ \forall x \in Vars$ gelten
$\implies [\lambda(v)]^{\#}(\sigma^{\#}) = \sigma^{\#'}$ mit $\sigma'(x)\triangle\sigma^{\#'}(x)\ \forall x \in Vars$
(Beweis durch Fallunterscheidung nach den möglichen Werten für $\lambda(v)$) $\square$

Weiterhin fordern wir, daß $\triangle$ die folgende Eigenschaft aufweist:

$$x \sqsubseteq y \wedge y\triangle y^{\#} \wedge y^{\#} \sqsubseteq z^{\#} \implies x\triangle z \;\; \forall x, y \in D,\ y^{\#}, z^{\#} \in D^{\#}$$

Eine Relation $\triangle$ mit dieser Eigenschaft bezeichnen wir als *Abstraktionsrelation.*

In unseren Beispielen hat $\triangle$ stets diese Eigenschaft.
Deshalb können wir für $P \in \{\text{SIGN,CONST}\}$ mit $P[v] = \bigsqcup_{\pi\in\Pi(v)} P[\pi]$ folgern:

$(\sigma_w, s) \xrightarrow{*} (\sigma, v) \implies \sigma(x)\triangle P[v](x)\ \forall x \in Vars.$

Angenommen, unsere Analyse liefere für den Programmpunkt v und die Variable x den Wert $d^{\#}$. Dann ist die Menge der konkreten Werte, die $d^{\#}$ 'abstrahiert', gegeben durch $\{d \mid d\triangle d^{\#}\} = \gamma(d^{\#})$, wobei $\gamma : D^{\#} \to P(D)$ eine monotone Abbildung ist mit $P(D) = \{X \subseteq D \mid x \in X,\ y \sqsubseteq x \implies y \in X\}$.
γ heißt *Konkretisierung.*

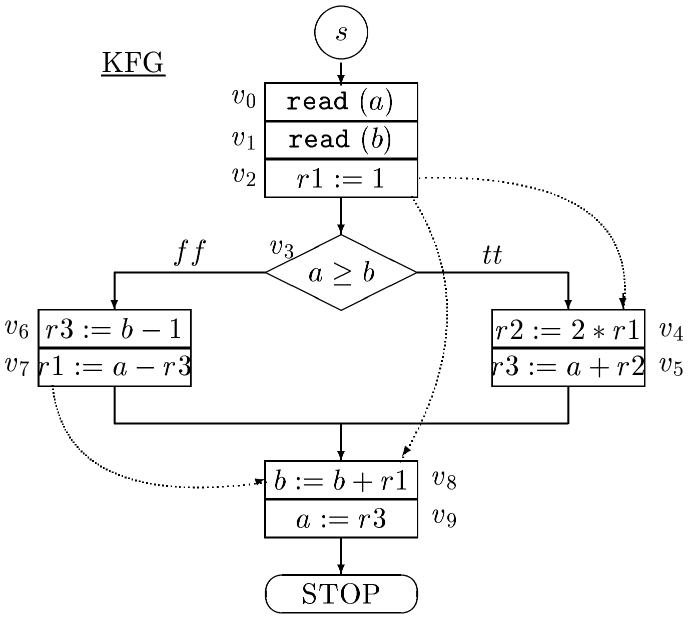
<u>Beispiele:</u>

- Vorzeichenanalyse: $\gamma(\top) = \mathbb{Z}_\bot,\ \gamma(0) = \{0, \bot\},\ \gamma(+) = \{z \in \mathbb{Z} \mid z > 0\} \cup \{\bot\},\ \gamma(-) = \{z \in \mathbb{Z} \mid z < 0\} \cup \{\bot\},\ \gamma(\bot) = \{\bot\}$
- Konstantenpropagation: $\gamma(\top) = \mathbb{Z}_\bot,\ \gamma(i) = \{i, \bot\}\ \forall i \in \mathbb{Z}_\bot,\ \gamma(\bot) = \{\bot\}$

Programmbeispiel für die Konstantenpropagation:

```
    begin
    var a, b, r1, r2, r3 :integer;
v0: read (a);
v1: read (b);
v2: r1 := 1;
v3: if (a >= b) then
v4:     r2 := 2 * r1;
v5:     r3 := a + r2
    else
v6:     r3 := b - 1;
v7:     r1 := a - r3
    fi;
v8:     b := b + r1;
v9:     a := r3
    end
```

KFG



Bei der Konstantenpropagation müssen wir hier die Werte für fünf Variablen verwalten, obwohl sich jeweils höchstens einer dieser Werte ändert !
Verbesserung: Berechne die Werte $\sigma^{\#}(x)$ nur dort, wo sie sich evtl. ändern (durch $\texttt{read}(x)$ oder $x := exp$) und propagiere diese Werte **direkt** zu den Knoten, die sie benutzen (solche Knoten also, die x im Ausdruck enthalten).

Für das Beispiel erhält man:

Knoten	Def. von	Benutzung von
v_0	a	$-$
v_1	b	$-$
v_2	$r1$	$-$
v_3	$-$	v_0, v_1
v_4	$r2$	v_2
v_5	$r3$	v_0, v_4
v_6	$r3$	v_1
v_7	$r1$	v_0, v_6
v_8	b	v_1, v_2, v_7
v_9	a	v_5, v_6

Sei $DEF[v]$ die Menge der **hinter** einem Knoten ankommenden Definitionen. $DEF[v],\ v \in V$ ist ein Vorwärtsproblem (Propagation der Information in Richtung der Kanten des KFG), gegeben durch den Verband $D := \mathcal{P}(\mathcal{D})$, wobei $\mathcal{D}_x := \{v \in V \mid v \text{ ist Definition von } x\}$ und $\mathcal{D} := \bigcup\limits_{x \in Vars} \mathcal{D}_x$.

Die Transferfunktionen $\mu(v)$ für Knoten v sind:

$\mu(v) = id$ für $\lambda(v) \in \{\texttt{nop}, \texttt{write}(exp), b\}$

$\mu(v)(A) = (A \backslash \mathcal{D}_x) \cup \{v\} = (A \cap \mathcal{D}_x^c) \cup \{v\}$ für $\lambda(v) \in \{\texttt{read}(x), x := exp\},\ A \in D$

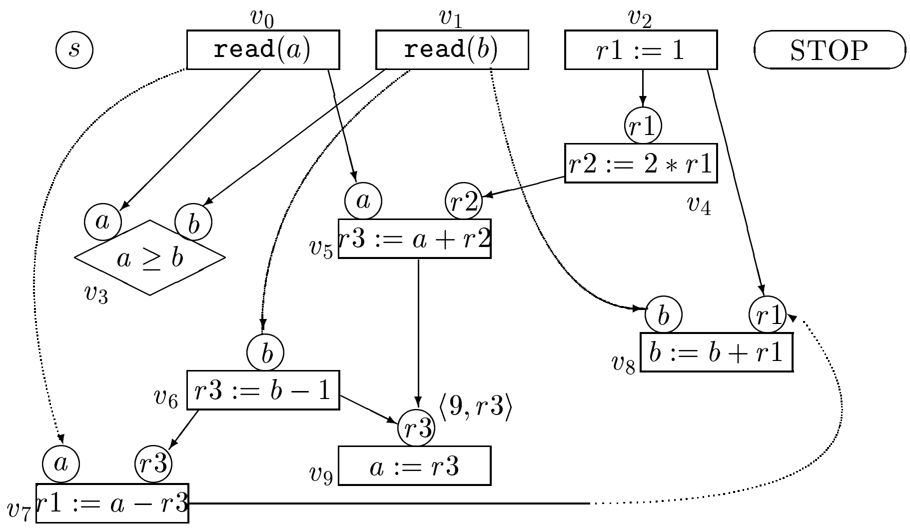
Der Startwert für s ist $DEF[s] = \emptyset$.
Eigentlich interessieren wir uns aber für die **vor** einem Knoten v ankommenden Definitionen. Und zwar für die Definitionen von Variablen x, auf deren Wert an v zugegriffen wird !
Sei $Var(v)$ die Menge von Variablen, die an v benutzt wird, d. h:

- $Var(v) = \emptyset$, falls $\lambda(v) \in \{\texttt{nop}, \texttt{read}(x), \text{START}, \text{STOP}\}$
- $Var(v) = Var(exp)$, falls $\lambda(v) \in \{exp, \texttt{write}(exp), x := exp\}$

Für $x \in Var(v),\ v \in V$ sei $DU[v,x] := \mathcal{D}_x \cap \bigcup_{(u,v)\in E} DEF[u]$ die Menge der **vor** v ankommenden Definitionen der Variablen x. Der (bipartite) Graph $DU_p = (VU_p, EU_p)$ für ein Programm p repräsentiert alle diese Abhängigkeiten von Benutzungen und Definitionen im KFG für p, wobei

$$VU_p := V \cup \{\langle v, x\rangle \mid x \in Var(v)\} \quad \text{und}$$
$$EU_p := \{(u, \langle v,x\rangle) \mid u \in DU[v,x]\} \cup \{(\langle v,x\rangle, v) \mid v \in V\} \quad \text{ist.}$$

DU_p für unser Programmbeispiel:



In DU_p genügt es offenbar, höchstens einen Wert pro Knoten zu verwalten und zu propagieren (nämlich denjenigen, der gerade definiert wird).
Insbesondere findet man im Beispiel vor Knoten 4, daß $r1 \equiv 1$ gilt. Folglich liefert $r2$ dort stets den Wert 2, und die rechte Seite in Knoten 5 kann ersetzt werden durch $a + 2$.

Gleichungssystem für Konstantenpropagation auf DU_p:

$$VAL[s] = \bot$$
$$VAL[\langle v,x\rangle] = \bigsqcup_{(w,\langle v,x\rangle)\in Eu_p} VAL[w]$$
$$VAL[v] = [exp]^{\#}[x \mapsto VAL[\langle v,x\rangle]],\text{ falls } \lambda(v) \in \{exp, \texttt{write}(exp), x := exp\}$$
$$VAL[v] = \top,\text{ falls } \lambda(v) = \texttt{read}(x)$$

Im Beispiel:

$$
\begin{array}{ll}
VAL[s] & = \bot \\
VAL[v_0] & = \top \\
VAL[v_1] & = \top \\
VAL[v_2] & = 1 \\
VAL[\langle v_3, a\rangle] & = VAL[v_0] \\
VAL[\langle v_3, b\rangle] & = VAL[v_1] \\
VAL[v_3] & = VAL[\langle v_3, a\rangle] \geq^{\#} VAL[\langle v_3, b\rangle] \\
VAL[\langle v_4, r1\rangle] & = VAL[v_2] \\
VAL[v_4] & = 2 *^{\#} VAL[\langle v_4, r1\rangle] \\
VAL[\langle v_5, a\rangle] & = VAL[v_0] \\
VAL[\langle v_5, r2\rangle] & = VAL[v_4] \\
VAL[v_5] & = VAL[v_0] +^{\#} VAL[v_4] \\
VAL[\langle v_6, b\rangle] & = VAL[v_1] \\
VAL[v_6] & = VAL[\langle v_6, b\rangle] -^{\#} 1 \\
VAL[\langle v_7, a, \rangle] & = VAL[v_0] \\
VAL[\langle v_7, r3\rangle] & = VAL[v_6] \\
VAL[v_7] & = VAL[\langle v_7, a, \rangle] -^{\#} VAL[\langle v_7, r3\rangle] \\
VAL[\langle v_8, b\rangle] & = VAL[v_1] \\
VAL[\langle v_8, r1\rangle] & = VAL[v_7] \sqcup VAL[v_2] \\
VAL[v_8] & = VAL[\langle v_8, b\rangle] +^{\#} VAL[\langle v_8, r1\rangle] \\
VAL[\langle v_9, r3\rangle] & = VAL[v_5] \sqcup VAL[v_6] \\
VAL[v_9] & = VAL[\langle v_9, r3\rangle]
\end{array}
$$

Die kleinste Lösung des Gleichungssystems liefert das gesuchte Ergebnis.

Satz:
Sei $(VAL[v])_{v\in V}$ und $(VAL[\langle v,x\rangle])_{\langle v,x\rangle\in DU_p}$ die kleinste Lösung des Gleichungssystems.
Dann gilt für jeden Knoten $v \in V$: $\lambda(v) \equiv x := exp \ \wedge \ CONST[v](x) = d \implies d \sqsubseteq VAL[v]$
Sei weiterhin w ein Vorgänger von v im KFG, dann gilt:
$\lambda(v) \in \{exp, \texttt{write}(exp), x := exp\} \wedge y \in Var(exp) \wedge CONST[w](y) = d \implies d \sqsubseteq VAL[\langle v, y\rangle]$

Auf den Beweis verzichten wir hier.

Aufwand zur Lösung des Gleichungssystems: $O(3 \cdot |DU_p|)$ (für den Worklist–Algorithmus)
Im schlimmsten Fall ist DU_p allerdings quadratisch in der Größe des Programms p. Dieser Worst–Case tritt aber i. a. nicht auf.
Falls DU_p doch 'zu groß' werden sollte, dann liegt der Grund darin, daß ein Knoten $\langle v, x\rangle$ sehr viele Vorgänger besitzen kann. In diesem Fall können wir ein neues Gleichungssystem aufstellen mit $VAL[\langle v,x\rangle] = \top$ (nun 'entfallen' die Vorgänger).
Für solche Knoten ist die kleinste Lösung dieses Systems immer noch korrekt, aber eventuell ungenauer.

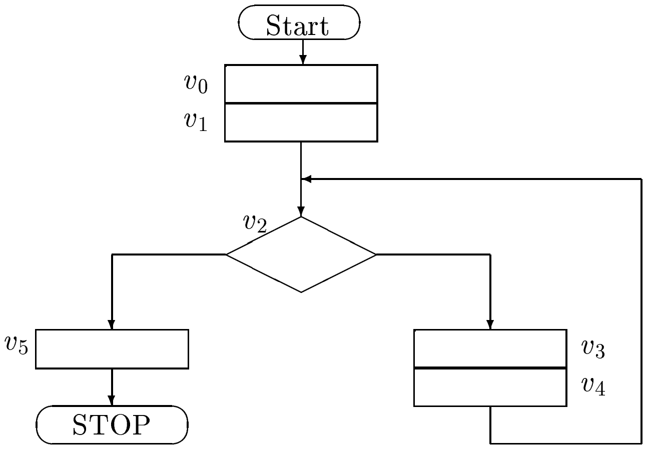
Fazit: Der Hauptaufwand bei der Konstantenpropagation besteht in der Konstruktion des Graphen DU_p, d. h. in der Bestimmung der ankommenden Definitionen.

2.6 Schnelle Algorithmen zur Lösung von Analyseproblemen

Beobachtung: Beim MOP sind die Mengen $\Pi(v)$ *reguläre* Teilmengen von V^*. $[e_v]$ bezeichne im weiteren die Menge der Pfade zum Knoten v, die durch den regulären Ausdruck e_v beschrieben wird.

Beispiel:

Knoten u	e_u mit $\Pi(u) = [e_u]$
v_0	v_0
v_1	$v_0\, v_1$
v_2	$v_0\, v_1 (v_2\, v_3\, v_4)^* v_2$
v_3	$v_0\, v_1 (v_2\, v_3\, v_4)^* v_2\, v_3$
v_4	$v_0\, v_1 (v_2\, v_3\, v_4)^* v_2\, v_3\, v_4$
v_5	$v_0\, v_1 (v_2\, v_3\, v_4)^* v_2\, v_5$



Definition:
Sei D ein vollständiger Verband, $\mathcal{F} \subseteq [D \to D]_c$ mit $id \in \mathcal{F}$ und $f, g \in \mathcal{F} \implies f \circ g \in \mathcal{F}$. Wir nehmen an, es gebe Operatoren $+ : \mathcal{F}^2 \to \mathcal{F}$ und $* : \mathcal{F} \to \mathcal{F}$ auf $\mathcal{F}$ mit den folgenden Eigenschaften:

(1) $+$ ist kommutativ und assoziativ;

(2) $id + id \sqsubseteq id$;

(3) $f_i \sqsubseteq f_1 + f_2$ für $i = 1, 2$;

(4) $id \sqsubseteq f^*$;

(5) $f \circ f^* \sqsubseteq f^*$ und $f^* \circ f \sqsubseteq f^*$.

Beispiel:
Sei $D := \mathcal{P}(A)$ und $\mathcal{F} := \{f : D \to D \mid \exists A_1, A_2 \subseteq A \text{ und } f(x) = (x \cap A_1) \cup A_2\}$, wobei A eine endliche Menge ist. Verwenden wir für $'+'$ den Operator $'\cup'$, für $'\sqsubseteq'$ die Relation $'\subseteq'$, dann erfüllt D mit $\mathcal{F}$ die oben aufgeführten Eigenschaften. Denn:
id ist die Funktion mit $A_1 = A$ und $A_2 = \emptyset$.
Sei $f_1(x) = (x \cap A_1) \cup A_2$ und $f_2(x) = (x \cap B_1) \cup B_2$. Dann gilt:
$(f_1 + f_2)(x) = ((x \cap A_1) \cup A_2) \cup ((x \cap B_1) \cup B_2) = (x \cap (A_1 \cup B_1)) \cup (A_2 \cup B_2) \in \mathcal{F}$
$(f_2 \circ f_1)(x) = (((x \cap A_1) \cup A_2) \cap B_1) \cup B_2 = (x \cap (A_1 \cap B_1)) \cup ((A_2 \cap B_1) \cup B_2) \in \mathcal{F}$

Sei $f(x) = (x \cap A_1) \cup A_2$ und $f^*(x) = x \cup A_2$
Es gilt (4), denn: $id(x) = (x \cap A) \cup \emptyset = x \subseteq x \cup A_2 = f^*(x)$
Es gilt (5), denn: $(f \circ f^*)(x) = ((x \cup A_2) \cap A_1) \cup A_2 = (x \cap A_1) \cup A_2 \subseteq x \cup A_2 = f^*(x)$

Beachte:

1. $f_1 \sqcup f_2 \sqsubseteq f_1 + f_2$
2. $f^n \sqsubseteq f^* \;\; \forall n > 0$
3. $f^2 \sqsubseteq id + f \implies \Box f := id + f$ erfüllt (4), (5) und ist damit als Definition für f^* geeignet.
   Beweis:
   (4) ist erfüllt, denn: $id \sqsubseteq id + f \overset{(3)}{=} \Box f$

(5) ist erfüllt, denn: $f \circ \Box f = f \circ (id + f) \sqsubseteq (id + f) \circ (id + f) \sqsubseteq id + (id + f) = (id + id) + f \sqsubseteq id + f = \Box f$

Beispiel:
Es sei $f(x) = (x \cap A_1) \cup A_2$. Dann gilt:
$f^2(x) = ((x \cap A_1) \cup A_2) \cap A_1) \cup A_2 = (x \cap A_1) \cup A_2 = f(x) \sqsubseteq (id + f)(x) = (x \cap (A_1 \cup A)) \cup (A_2 \cup \emptyset) = x \cup A_2$

Definition:
Sei $\mu : V \longrightarrow \mathcal{F}$ Abbildung (V Knotenmenge des KFG). μ läßt sich zu einer Abbildung $\mu : R_V \longrightarrow \mathcal{F}$ erweitern, wobei R_V die Menge aller regulären Ausdrücke über V (ohne $\emptyset$) ist, durch

- $\mu(\varepsilon) := id$
- $\mu(e_1 + e_2) := \mu(e_1) + \mu(e_2)$
- $\mu(e_1 \cdot e_2) := \mu(e_2) \circ \mu(e_1)$
- $\mu(e^*) := \mu(e)^*$

Satz: Für alle $e \in R_V$ gilt:

(1) $\mu(\pi) \sqsubseteq \mu(e) \;\; \forall \pi \in [e]$ d. h. $\bigsqcup_{\pi \in [e]} \mu(\pi) \sqsubseteq \mu(e)$

(2) Ist jede Funktion $f \in \mathcal{F}$ distributiv, dann gilt sogar:

$$\bigsqcup_{\pi \in [e]} \mu(\pi) = \mu(e),$$

sofern $+ \equiv \sqcup$ und $f^* = \bigcup_{n \geq 0} f^n$.

Beweis: Induktion über die Struktur von e $\Box$

Folgerung:
Unter den folgenden Voraussetzungen können wir damit Analyseprobleme schneller lösen:

- Für jedes $v \in V$ kann 'schnell' ein regulärer Ausdruck e_v gefunden werden mit $[e_v] = \Pi(v)$.
- Die Operatoren $+$, $\circ$, $*$ sind effizient berechenbar (wie es im Beispiel der Fall war).

Diese regulären Ausdrücke können für 'reduzierbare' Graphen in $\mathrm{O}(n \cdot \alpha(n, n))$ berechnet werden (Tarjan), wobei $n = |\mathrm{KFG}|$, α inverse Ackermannfunktion.
Lineare Algorithmen existieren für blockstrukturierte Programme (wie wir sie hier betrachten).
Dabei berechnen wir in Wirklichkeit nicht **zuerst** für jeden Knoten v den Ausdruck e_v, und werten diesen **dann** in 2. Phase aus, sondern berechnen **gleich** $\mu(e_v)$, $v \in V$.

Ziel: Bestimme für jedes v den regulären Ausdruck e_v für $\Pi(v)$. Besser gesagt: bestimme $\mu(e_v)$.

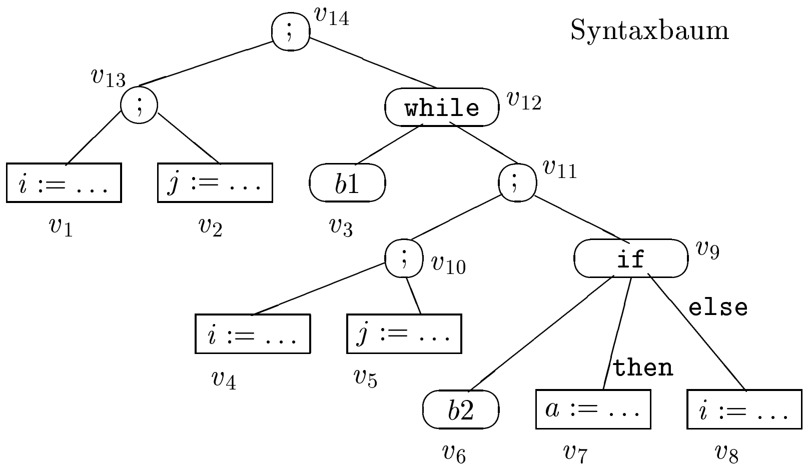
Idee: Wir gehen in zwei Phasen vor.

1. Setze μ fort auf Statements und Statementfolgen.
2. Sammle die Effekte auf, die zu Statements vor einem Knoten gehören.

Die Implementierung erfolgt direkt auf dem Syntaxbaum des Programms.

Beispiel: ankommende Definitionen ($DEF[v]$)

```
begin
var ...
  ⋮
i := ... ;
j := ... ;
while b1 do
  i := ... ;
  j := ... ;
  if b2 then
    a := ... ;
  else
    i := ... ;
  fi
od ;
  ⋮
end
```



$D = \mathcal{P}(\mathcal{D}), \quad \mathcal{D} = \{v_1, v_2, v_4, v_5, v_7, v_8\} \quad \mu(v) = (x \cap A) \cup B = f(x)$

Knoten	A	B	$f(x)$
v_1	$\{v_2, v_5, v_7\}$	$\{v_1\}$	$(x \cap \{v_2, v_5, v_7\}) \cup \{v_1\}$
v_2	$\{v_1, v_4, v_7, v_8\}$	$\{v_2\}$	.
v_3	$\{v_1, v_2, v_4, v_5, v_7, v_8\}$	$\emptyset$	.
v_4	$\{v_2, v_5, v_7\}$	$\{v_4\}$	.
v_5	$\{v_1, v_4, v_7, v_8\}$	$\{v_5\}$	$(x \cap \{v_1, v_4, v_7, v_8\}) \cup \{v_5\}$
v_6	$\{v_1, v_2, v_4, v_5, v_7, v_8\}$	$\emptyset$	.
v_7	$\{v_1, v_2, v_4, v_5, v_8\}$	$\{v_7\}$	.
v_8	$\{v_2, v_5, v_7\}$	$\{v_8\}$	.

1. Phase:

$s \equiv$ `if` b `then` s_1 `else` s_2 `fi`, dann gilt: $\mu(s) := (\mu(s_1) + \mu(s_2)) \circ \mu(b)$
$s \equiv$ `while` b `do` s' `od`, dann gilt: $\mu(s) := (\mu(b) \circ \mu(s'))^* \circ \mu(b)$
$s \equiv s_1; s_2$ dann gilt $\mu(s) := \mu(s_2) \circ \mu(s_1)$

Im Beispiel oben:

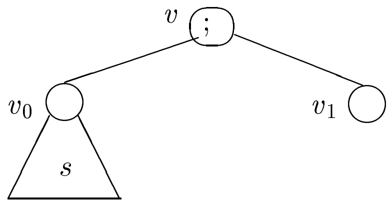
$$\begin{aligned}
\mu(v_9) &= \mu(v_7) + \mu(v_8) \\
\mu(v_{10}) &= \mu(v_5) \circ \mu(v_4) \\
\mu(v_{11}) &= \mu(v_9) \circ \mu(v_{10}) \\
\mu(v_{12}) &= \mu(v_{11})^* \\
\mu(v_{13}) &= \mu(v_2) \circ \mu(v_1) \\
\mu(v_{14}) &= \mu(v_{12}) \circ \mu(v_{13})
\end{aligned}$$

Diese Werte können in einem Pass bottom–up berechnet werden.

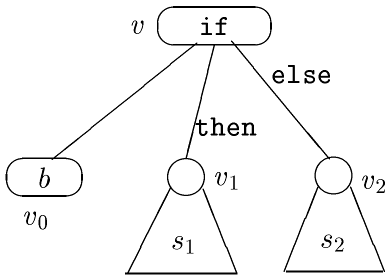
2. Phase:

Wir sammeln die Effekte **vor** Erreichen eines Statements *stat* top–down auf (r sei die Wurzel des Syntaxbaums).

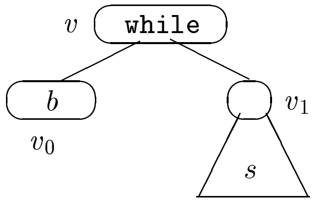
$$\beta(r) = d$$



$$\beta(v_0) = \beta(v)$$
$$\beta(v_1) = \mu(s)(\beta(v))$$



$$\beta(v_0) = \beta(v)$$
$$\beta(v_1) = \mu(b)(\beta(v))$$
$$\beta(v_2) = \mu(b)(\beta(v))$$



$$\beta(v_0) = (\mu(s) \circ \mu(b))^*(\beta(v))$$
$$\beta(v_1) = \mu(b)(\beta(v_0))$$

Fortsetzung unseres Beispiels:

$$\begin{aligned}
\beta(v_{14}) &= \emptyset \\
\beta(v_{13}) &= \beta(v_{14}) \\
\beta(v_1) &= \beta(v_{13}) \\
\beta(v_2) &= \mu(v_1)(\beta(v_{13})) \\
\beta(v_{12}) &= \mu(v_{13})(\beta(v_{14})) \\
\beta(v_3) &= (\mu(v_{11}))^*(\beta(v_{12})) \\
\beta(v_{11}) &= \beta(v_3)
\end{aligned}
\qquad
\begin{aligned}
\beta(v_{10}) &= \beta(v_{11}) \\
\beta(v_4) &= \beta(v_{10}) \\
\beta(v_5) &= \mu(v_4)(\beta(v_{10})) \\
\beta(v_9) &= \mu(v_{10})(\beta(v_{11})) \\
\beta(v_6) &= \beta(v_9) \\
\beta(v_7) &= \beta(v_9) \\
\beta(v_8) &= \beta(v_9)
\end{aligned}$$

Das gesuchte Ergebnis für ein Blatt v im Syntaxbaum ist dann $(\mu(e_v))(d) = \mu(v)(\beta(v))$.
Für die Realisierung des Algorithmus' benötigt man effiziente Implementierungen der Operatoren $+$, $*$, $\circ$ auf $\mathcal{F}$, in unserem Beispiel Definitionen von Mengenoperationen $\cap$ und $\cup$. Dies geschieht i. a. mit Hilfe von Bitvektoren (effizient, sofern die behandelten Mengen nicht zu groß sind, d. h. $\sharp Menge \leq 320$).

Nicht behandelte Probleme:

- Untersuchung von abstrakten Bereichen ohne acc (z. B. Intervalle);
- Pointeranalyse.

Kapitel 3

Analyse funktionaler Programme

Beispiele für funktionale Programmiersprachen:

- λ–Kalkül;
- LISP;
- ML;
- Miranda^{TM};
- Haskell 1.3.

3.1 Einführung

Programme arbeiten auf Datenstrukturen.

Übliche Datenstrukturen funktionaler Programme:

- Aufzählungen: `bool == True + False`
  `week == Monday +···+ Sunday`
- Listen: `list t == [] + (t:list t)`
  Abkürzung: `(x1:(x2:(x3:[])))` $\equiv$ `[x1,x2,x3]`
- Bäume: `tree t == Leaf t + Node (tree t) (tree t)`

'`:`' bezeichnet man als zweistelligen Konstruktor (in Infixschreibweise), `True` und `False` als nullstellige Konstruktoren.

Übliche Funktionen funktionaler Programme:

- `if True x y = x;   if False x y = y`
  etwa: `(((if True) 1) 2)` $\equiv$ `if True 1 2` $\to$ `1`
  `(((if False) 1) 2)` $\equiv$ `if False 1 2` $\to$ `2`
  Eine Funktion wird mithilfe von Fallunterscheidungen (getrennt durch Semikolon) definiert. Die Auswahl des Falles geschieht durch Vergleich der aktuellen Parameter mit sogenannten Patterns in den linken Seiten.

- `first (x:xs) = x`
  etwa: `first [1,2,3]` $\xrightarrow{(*)}$ `1`   `(*) x`$\mapsto$`1  und  xs`$\mapsto$`[2,3]`
  Patterns können Variablen enthalten. Diese werden beim Vergleichen des Patterns mit den aktuellen Parametern gebunden.

- `app [] y = y;   app (x:xs) y = x:(app xs y)`   (wobei `y` Liste)
  etwa: `app [1,2] [3]` $\rightarrow$ `1:(app [2] [3])` $\rightarrow$ `1:(2:(app [] [3]))`
  $\rightarrow$ `1:(2:[3]) = [1,2,3]`

- `rev [] = []   rev (x:xs) = app (rev xs) [x]`
  etwa: `rev [1,2,3]` $\rightarrow$ `app (rev [2,3]) [1]`
  $\rightarrow$ `app (app (rev [3]) [2]) [1]`
  $\rightarrow$ `app (app (app (rev []) [3])[2]) [1]`
  $\rightarrow$ `app (app (app [] [3])[2]) [1]`
  $\rightarrow$ `app (app [3] [2]) [1]`
  $\rightarrow \cdots \rightarrow$ `app [3,2] [1]` $\rightarrow \cdots \rightarrow$ `[3,2,1]`

- höhere Funktionen:
  `map f [] = [];   map f (x:xs) = (f x):(map f xs)`
  `map` wendet die Funktion `f` auf alle Elemente einer Liste an.

- `fold f a [] = a;   fold f a (x:xs) = f x (fold f a xs)`

Beispiele:

- `rev = fold addlast []`, wobei `addlast x l = app l [x]`
  etwa: `rev [1,2]` $\rightarrow$ `fold addlast [] [1,2]`
  $\rightarrow$ `addlast 1 (fold addlast [] [2])`
  $\rightarrow$ `addlast 1 (addlast 2 (fold addlast [] []))`
  $\rightarrow$ `addlast 1 (addlast 2 [])`
  $\rightarrow$ `addlast 1 [2]` $\rightarrow$ `[2,1]`

- `sosq l = fold (+) 0 (map square l)`, wobei `square x = x*x`
  (`sosq` bildet die Summe über die Quadrate der Elemente einer Integerliste)
  etwa: `sosq [3,4]` $\rightarrow$ `fold (+) 0 (map square [3,4])`
  $\rightarrow$ `fold (+) 0 ((square 3):(map square [4]))`
  $\stackrel{*}{\rightarrow}$ `fold (+) 0 (9:(map square [4]))` $(*)$
  $\rightarrow$ `fold (+) 0 (9:((square 4):(map square [])))`
  $\stackrel{*}{\rightarrow}$ `fold (+) 0 (9:(16:(map square [])))` $(*)$
  $\rightarrow$ `fold (+) 0 [9,16]`
  $\rightarrow$ `9 + (fold (+) 0 [16])`
  $\rightarrow$ `9 + (16 + (fold (+) 0 []))`
  $\rightarrow$ `9 + (16 + 0)` $\rightarrow$ `9 + 16` $\rightarrow$ `25`

Beachte: An den Stellen $(*)$ hätten wir die Funktionsaufrufe von `square` auch nach einer anderen Strategie auswerten können. Die hier verwendete Strategie heißt *Call by Value* (CBV) oder *left–most–inner –most*.
Im nächsten Beispiel betrachten wir die *Call by Name*–Strategie (CBN) oder *left–most–outer–most*.

Beispiel:

```
sosq [3,4]  →  fold (+) 0 (map square [3,4])
            →  fold (+) 0 ((square 3):(map square [4]))
            →  (+) (square 3) (fold (+) 0 (map square [4]))
            →  (square 3) + (fold (+) 0 (map square [4]))
            →  9 + (fold (+) 0 (map square [4]))
            →  9 + (fold (+) 0 (square 4):(map square []))
            →  9 + ((square 4) + (fold (+) 0 (map square [])))
            →  9 + (16 + (fold (+) 0 (map square [])))
            →  9 + (16 + (fold (+) 0 []))
            →  9 + (16 + 0) → 9 + 16 → 25
```

Sofern die Berechnung terminiert, kommt es nicht auf die Reihenfolge der Auswertungsschritte an ! Deshalb soll man jeweils die günstigste Strategie wählen. Aber welche ist nun die günstigste Strategie ?

Beobachtung:

- CBV wertet erst **alle** Argumente aus, bevor ein Funktionsaufruf durchgeführt wird. Allerdings ist die Auswertung aller Argumente nicht immer erwünscht (z. B. bei `if`), da dies eventuell zu Nichtterminierungen führt (z. B. bei Rekursion) !
  (Ein Ausweg wäre hier, ein 'eingebautes' `if` einzuführen, das entgegen der generellen Strategie zuerst nur die Bedingung auswertet.)

- CBN verzögert die Auswertung von Funktionsaufrufen so lange wie möglich (sog. 'lazyness'), und läuft deshalb nicht in die 'Nichtterminierungsfalle' wie bei CBV.
  Das Verzögern der Auswertung kann aber gravierende Nachteile haben, denn:
  1. ein später auszuwertender Teilausdruck wird eventuell vor der Auswertung kopiert. Dies führt dazu, daß eine mehrfache Auswertung erforderlich ist. Ein Ausweg wäre hier, das Sharing von mehrfach zu verwendenden Teilausdrücken einzuführen.
  2. Das Aufheben von unausgewerteten Ausdrücken zur späteren Verwendung führt zu erhöhtem Verwaltungsaufwand für die Speicherung und für Tests, die bei jedem Zugriff auf ein Objekt prüfen, ob es bereits ausgewertet wurde, oder nicht.

Idee: Kombiniere die Vorteile von CBN (Terminierung) und die Effizienz von CBV miteinander. Dazu wird eine Analyse durchgeführt, die überprüft, ob die Argumente einer Funktion problemlos vorneweg ausgewertet werden können (die sog. *Striktheitsanalyse*). Liefert diese Analyse eine 'Unbedenklichkeitsbescheinigung' für ein Argument, so wird dieses *by value* übergeben, d. h. vor dem Funktionsaufruf ausgewertet.

Weitere interessante Fragen:

- An welche anderen Ausdrücke kann eine Variable im Laufe der Programmausführung gebunden sein ? Insbesondere würde man gerne erfahren, an welche Funktion funktionale Argumente gebunden sein können. Dies ist nützlich, um beispielsweise den Overhead höherer Funktionen zu eliminieren, oder um in geschickter Weise Register zu vergeben. Die hierzu notwendige Analyse ist die sog. *Kontrollflußanalyse*.

- Gibt es mehr als einen Konsumenten einer Datenstruktur, dann kann diese nicht 'in place' modifiziert werden, sondern vielmehr muß dann ein Teil der Datenstruktur kopiert werden. Dies führt zur *Sharinganalyse*.

Ein weiteres Konzept funktionaler Sprachen:

Zusammenbinden von Funktionen und einem Ausdruck, in dem sie benutzt werden können:

```
letrec id [] = [];  id (x:xs) = x:(id xs) in id (id [1,2,3])
```

Wert = Wert des Hauptausdrucks
Scope von `id` = Hauptausdruck + alle rechten Seiten in `letrec`.

Wir fassen nun zusammen.

3.2 Die Sprache

Die Grammatik für funktionale Programme:

pr	$::=$	$e \mid def; pr$	(Programm)
def	$::=$	$t\ ==\ cdefs$	(Datentypdefinition)
$cdefs$	$::=$	$calt \mid calt\ +\ cdefs$	(Konstruktordefinitionen)
$calt$	$::=$	$C\ t_1 \dots t_n \quad (n \geq 0)$	(Konstruktor)
e	$::=$	C	(0–stelliger Konstruktor)
		$\mid (C\ e_1 \dots e_n)$	(n–stelliger Konstruktor)
		$\mid (e_1\ e_2)$	(Applikation)
		$\mid x$	(Variable)
		$\mid (\texttt{letrec}\ fdefs\ \texttt{in}\ e_0)$	
$fdefs$	$::=$	$falt \mid falt; fdefs$	(Funktionsdefinitionen)
$falt$	$::=$	$f\ p\ =\ e$	(Funktionsalternative)
p	$::=$	$C \mid x \mid (C\ p_1 \dots p_n)$	(Pattern)

Bemerkungen:

(1) Funktionen sind nur einstellig. Das ist aber keine Einschränkung (Übung) !

(2) Es gibt keine Integers, keine eingebauten Funktionen und keine Prädikate (wie beispielsweise $\leq$, $<$, $=$ usw.) darauf; diese können aber leicht hinzugefügt werden (Übung).

(3) Patterns enthalten unter Umständen Variablen. Wir fordern, daß keine Variable zweimal in einem Pattern auftaucht (*lineare Patterns*).

Wir treffen weitere Einschränkungen der Sprache, damit das weitere 'einfacher' zu beschreiben ist:

(a) Patterns haben nur eine der folgenden Formen: $p\ ::=\ C \mid x \mid (C\ x_1 \dots x_n)$

(b) Jedes `letrec` definiert genau eine Funktion. Dies ermöglicht die Einführung einer vereinfachten Notation:
   - Für $f\ x\ =\ e$ schreiben wir $f\ =\ (\lambda x.e)$ (λx heißt formaler Parameter, e Rumpf)
   - Für $f\ p_1\ =\ e_1;\ \dots;\ f\ p_n\ =\ e_n$ schreiben wir:
     $f\ =\ \lambda x.(\texttt{case}\ x\ \texttt{of}\ p_1 \to e_1;\ \dots;\ p_n \to e_n)$
     (Wir schreiben auch abkürzend: $f\ =\ \lambda x.(\texttt{case}\ x\ \texttt{of}\ p_1 \to e_1;\ altlist)$,
     wobei *altlist* die Alternativen $p_2 \to e_2;\ \dots;\ p_n \to e_n$ umfaßt)

Beispiel:

$\texttt{first = }\lambda\texttt{l.(case l of (x:xs)}\rightarrow\texttt{x)}$
$\texttt{rev = }\lambda\texttt{l.(case l of []}\rightarrow\texttt{[]; (x:xs)}\rightarrow\texttt{app (rev xs) [x])}$

Rekursive Definitionen notieren wir mithilfe des einstelligen Operators μ:
$\texttt{letrec f} = e_1 \texttt{ in } e_0 \quad \Longrightarrow \quad \texttt{let f} = \mu\texttt{f}'.e_1[\texttt{f}'/\texttt{f}] \texttt{ in } e_0,$
wobei `f'` ein Funktionsname ist, der noch nicht in e_0 vorkommt.

$[\texttt{f}'/\texttt{f}]$ bezeichnet die Sustituierung aller freien Vorkommen von `f` durch `f'` und `let` eine **nicht-rekursive** Variablendefinition. μ heißt *Fixpunktoperator*.

Beispiel:

$\texttt{rev = }\mu\texttt{r.(}\lambda\texttt{l.(case l of []}\rightarrow\texttt{[]; (x:xs)}\rightarrow\texttt{app (r xs) [x]))}$

Damit erhalten wir eine modifizierte Grammatik für Ausdrücke, mit:

$$
\begin{array}{lll}
e ::= & C & \text{(0–stelliger Konstruktor)} \\
 & \mid (C\ e_1 \ldots e_n) & (n\text{–stelliger Konstruktor}) \\
 & \mid (e_1\, e_2) & \text{(Applikation)} \\
 & \mid x & \text{(Variable)} \\
 & \mid (\lambda x.e) & \text{(Abstraktion)} \\
 & \mid (\texttt{case } e_0 \texttt{ of } p_1 \rightarrow e_1; \ldots; p_k \rightarrow e_k) & \text{(Auswahl)} \\
 & \mid (\mu x.e) & \text{(Rekursion)} \\
 & \mid (\texttt{let } x = e_1 \texttt{ in } e_0) & \text{(nichtrek. Definition)}
\end{array}
$$

Funktionsdefinitionen und Funktionsalternativen unserer ursprünglichen Grammatik entfallen nun.

3.3 Operationelle Semantik

Wir definieren Auswertungsrelationen '$\rightarrow$' und '$\Rightarrow$' auf Termen.

1. Ziel:
Auswertung zu 'partiellen Werten', d. h. Ausdrücken der Form:
$w ::= C \mid (C\ e_1 \ldots e_n) \mid (\lambda x.e)$

Um die Auswertungsrelation '$\rightarrow$' zu partiellen Werten zu definieren, gehen wir in mehreren Schritten vor.

1) Berechnungsschritte

$$
\begin{array}{ll}
\texttt{let}: & (\texttt{let } x = e_1 \texttt{ in } e_0) \rightarrow e_0[e_1/x] \\
\beta: & ((\lambda x.e_0)\ e_1) \rightarrow e_0[e_1/x] \\
\texttt{case}: & (\texttt{case } e_0 \texttt{ of } x \rightarrow e_1; \mathit{altlist}) \rightarrow e_1[e_0/x] \\
 & (\texttt{case } (C\, d_1 \ldots d_n) \texttt{ of } C\, x_1 \ldots x_n \rightarrow e_1; \mathit{altlist}) \rightarrow e_1[d_1/x_1, \ldots d_n/x_n] \\
 & (\texttt{case } (C\, d_1 \ldots d_n) \texttt{ of } C'\, x_1 \ldots x_n \rightarrow e_1; \mathit{altlist}) \\
 & \qquad\qquad \rightarrow (\texttt{case } (C\, d_1 \ldots d_n) \texttt{ of } \mathit{altlist}) \text{ für } C \neq C' \\
\mu: & (\mu x.e) \rightarrow e[(\mu x.e)/x]
\end{array}
$$

2) Auswertungskontexte

Zu jedem Term, der kein partieller Wert ist, müssen wir denjenigen Teilterm identifizieren, wo ein Berechnungsschritt stattfinden kann. Wir betrachten hier nur CBN.
Wir definieren die Menge der *Auswertungskontexte* für die CBN–Strategie durch:

$$\begin{array}{lll} k & ::= & \circ \\ & & \mid (k\ e) \\ & & \mid (\texttt{case}\ k\ \texttt{of}\ C\,x_1 \ldots x_n\ \rightarrow e; \mathit{altlist}) \end{array}$$

Jeder Kontext enthält genau ein Vorkommen von $\circ$. Für $k[e/\circ]$ schreiben wir einfach $k[e]$.

Satz:
Jeder Ausdruck e, der kein partieller Wert ist, läßt sich eindeutig zerlegen in $e = k[r/\circ]$, wobei r eine der folgenden Formen besitzt:

$$\begin{array}{lll} r & ::= & \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_0 \\ & & \mid (w\ e) \\ & & \mid \texttt{case}\ w\ \texttt{of}\ C\,x_1 \ldots x_n\ \rightarrow e_1; \mathit{altlist} \\ & & \mid \texttt{case}\ e\ \texttt{of}\ x \rightarrow e_1; \mathit{altlist} \\ & & \mid \texttt{case}\ e\ \texttt{of} \qquad (\star) \\ & & \mid \mu x.e \end{array}$$

($\star$) Solche 'Fehlerzustände' treten auf, falls die Liste der Alternativen in `case` leer ist, oder ein partieller Wert in der Funktionsposition einer Applikation auftritt, der **keine** Funktion darstellt (z. B. `1` oder `True`).

Beispiele:

Sei $e :=$ (`let first = λl.case l of (x:xs)→x in first [1,2]`)
Für e ist $e = k[r]$ mit $k = \circ$ und $r = e$.
Nach einem `let`–Berechnungsschritt erhalten wir mit
`(first [1,2])`[`(λl.case l of (x:xs)→x)` / `first`]
den Ausdruck e_1 = (`(λl.case l of (x:xs)→x) [1,2]`)
mit $e_1 = k_1[r_1]$ für $k_1 = \circ$, $r_1 = e_1$.
Nach einem β–Berechnungsschritt erhalten wir mit
`(λl.case l of (x:xs)→x)`[`[1,2]` / `l`]
den Ausdruck e_2 = `(case (1:(2:[])) of (x:xs)→x)`
mit $e_2 = k_2[r_2]$ für $k_2 = \circ$, $r_2 = e_2$.
Ein `case`–Berechnungsschritt liefert letztendlich mit
`(case (1:(2:[])) of (x:(x':xs))→x)`[`1` / `x`]
den Wert `1`.

Nun ein etwas komplexerer Ausdruck:

```
Sei e :=  (let first = λl.case l of (x:xs)→x
            in let rest = λl'.case l' of (y:ys)→ys
                in (first (rest [1,2])))
```

Wir führen nun Berechnungen auf e aus:

- `let`–Berechnungsschritt:

```
e1  =  (let rest = λl'.case l' of (y:ys)→ys
           in ((λl.case l of (x:xs)→x) (rest [1,2])))
```

Für e_1 ist $e_1 = k_1[r_1]$ mit $k_1 = \circ$, $r_1 = e_1$

- `let`–Berechnungsschritt:
  $e_2 =$ `((`λ`l.case l of (x:xs)`$\rightarrow$`x) ((`λ`l' case l' of (y:ys)`$\rightarrow$`ys) [1,2]))`
  Für e_2 ist $e_2 = k_2[r_2]$ mit $k_2 = \circ$, $r_2 = e_2$

- β–Berechnungsschritt:
  $e_3 =$ `(case ((`λ`l'.case l'of (y:ys)`$\rightarrow$`ys) [1,2]) of (x:xs)`$\rightarrow$`x)`
  Für e_3 ist $e_3 = k_3[r_3]$ mit $k_3 =$ `case` $\circ$ `of` $(\texttt{x}:\texttt{xs}) \rightarrow \texttt{x}$
  und r_3=`(`λ`l'.case l' of (y:ys)` $\rightarrow$ `ys) [1,2]`
  Die Berechnung wird nun fortgesetzt mit dem Teilausdruck r_3.

3) Terme

Für beliebige Terme definieren wir nun $e_1 \rightarrow e_2$, falls $e_1 = k[r]$ und $e_2 = k[e]$ mit $r \rightarrow e$ gilt, wobei $r \rightarrow e$ ein Berechnungsschritt ist gemäß 1).

2. Ziel:
Wir definieren schließlich eine Auswertungsrelation '$\Rightarrow$', die Terme zu *vollständigen Werten* auswertet.
Vollständige Werte sind: $v \ ::= \ C \mid (C\ v_1 \ldots v_k) \mid (\lambda x.e)$

Vorgehensweise:
a) Die Auswertung darf nicht bei Erscheinen des ersten Konstruktors an der Wurzel stoppen.
Dazu definieren wir die Menge der *aktiven Kontexte* durch: $a \ ::= \ \circ \mid (C\ v_1 \ldots v_{k-1}\, a\, e_{k+1} \ldots e_n)$

Satz:
Jeder Ausdruck e, der kein vollständiger Wert ist, läßt sich zerlegen zu $e = a[k[r]]$.

Beispiel:

$e =$ `[1,2,3]` Wert
$e =$ `[(`λ`x.x) 1,2]` $= a[k[r]]$ mit $a =$ `[`$\circ$`,2] = (`$\circ$`:2)`, $k = \circ$ und $r =$ `(`λ`x.x) 1`
$e =$ `[1,(((`λ`x.x)(`λ`y.y)) 2)]` $= a[k[r]]$ mit $a =$ `[1,`$\circ$`]=(1:`$\circ$`)`, $k =$ `(`$\circ$`,2)` und
$r =$ `(`λ`x.x)(`λ`y.y)`

b) Damit definieren wir $e_1 \Rightarrow e_2$, falls $e_1 = a[k[r]]$ und $e_2 = a[k[e]]$ und $r \rightarrow e$ gilt.
Der Wert eines Ausdrucks e ergibt sich damit zu $[e] = v$ sofern $e \overset{*}{\Rightarrow} r$ (v vollständiger Wert).

3.4 Kontrollflußanalyse

Die Kontrollpunkte eines Programms *pr* sind die Teilausdrücke des Programms.
Sei E_{pr} die Menge aller dieser Teilausdrücke.
Wir suchen nun für jeden Teilausdruck e die Menge von Teilausdrücken, so daß die Auswertung von e die Auswertung von Ausdrücken in dieser Menge erfordert, wobei deren Resultate zu einem Resultat für e führen.

Beispiel:

```
let first = λl.case l of (x:xs)→x
  in let rest = λl'.case l' of (y:ys)→ys
    in first (rest [1])
```

Auswertung von	führt zur Auswertung von
`rest [1]`	`ys`
`l'`	`[1]`
`y`	`1`
`ys`	`[]`
`first (rest [1])`	`x`
`l`	`rest [1]`

Idee: Wir stellen ein Ungleichungssystem auf !
Als abstrakten Bereich verwenden wir $\mathcal{P}(E_{pr})$. Die Variablen des Ungleichungssystems seien $[e]^\#$ mit $e \in E_{pr}$, die jeweils die Menge aller Teilausdrücke von e als Wert enthalten, die zur Auswertung eben dieses Ausdrucks e benötigt werden.

Konstruktion des Ungleichungssystems S_E für ein Programm *pr*

Für jeden Teilausdruck e erhalten wir eine Ungleichung $[e]^\# \supseteq \{e\}$. Weitere Ungleichungen erhalten wir folgendermaßen:

- $e \equiv \texttt{let}\ x = e_1\ \texttt{in}\ e_0 \;\Rightarrow\; [e]^\# \supseteq [e_0]^\#$ und $[x]^\# \supseteq [e_1]^\#$
- $e \equiv \mu x.e_0 \;\Rightarrow\; [e]^\# \supseteq [e_0]^\#$ und $[x]^\# \supseteq [e]^\#$
- $e \equiv (e_1\ e_2) \;\Rightarrow\; [e]^\# \supseteq (\{\lambda x.e'\} \cap [e_1]^\#); [e']^\#$ und $[x]^\# \supseteq (\{\lambda x.e'\} \cap [e_1]^\#); [e_2]^\#$
- $e \equiv (\texttt{case}\ e_0\ \texttt{of}\ \mathit{altlist}) \;\Rightarrow$
$$\Rightarrow \left\{ \begin{array}{l} [e]^\# \supseteq (\{C\ d_1 \ldots d_n\} \cap [e_0]^\#); [e_1]^\# \\ [x_i]^\# \supseteq (\{C\ d_1 \ldots d_n\} \cap [e_0]^\#); [d_i]^\# \end{array} \right\} \text{falls } C\ x_1 \ldots x_n \to e_1 \text{ in } \mathit{altlist},$$
$$\Rightarrow \{\ [e]^\# \supseteq [e_1]^\# \text{ und } [x]^\# \supseteq [e_0]^\#\ \} \text{ falls } x \to e_1 \text{ in } \mathit{altlist}$$

Zur Erinnerung: ';' ist der Operator mit
$$;: \mathcal{P}(M) \times \mathcal{P}(M) \longrightarrow \mathcal{P}(M) \text{ ist definiert durch } A;B = \begin{cases} \emptyset, & \text{falls } A = \emptyset \\ B, & \text{sonst} \end{cases}$$

Probleme:

1. Wie löst man ein Ungleichungssystem ?
2. Die Größe dieses Ungleichungssystems ist $O(|pr|^2)$, wobei $|pr| = \sharp E_{pr}$. Wir müssen also ein sehr großes Ungleichungssystem lösen.
3. Erhalten wir eine korrekte Lösung ?

Wir wollen nun eine Antwort für Problem 1. finden.

3.4.1 Ungleichungssysteme auf vollständigen Verbänden

Definition: Sei D ein vollständiger Verband und S ein endliches System von Ungleichungen der Form $x \sqsupseteq exp$, wobei exp eine stetige Funktion $[exp] : (Vars \to D) \longrightarrow D$ beschreibt ($Vars$ sei die Menge der Variablen des UGS).

Dann heißt $\eta : Vars \to D$ ein *Modell* für S, falls für alle Ungleichungen $x \sqsupseteq exp$ in S gilt: $\eta(x) \sqsupseteq [exp](\eta)$.

Zu S konstruieren wir ein Gleichungssystem S' mit $x = f(x)$, $x \in Vars$, wobei $f(x) = e_1 \sqcup \cdots \sqcup e_k$ gilt, falls $\{e_1, \ldots, e_k\} = \{exp \mid x \sqsupseteq exp \text{ istUngleichungin} S\}$.

Satz: Das Ungleichungssystem S besitzt ein eindeutig bestimmtes kleinstes Modell $\bar{\eta}$. Dieses Modell $\bar{\eta}$ ist die kleinste Lösung des Gleichungssystems S'.

Beweis:
Offenbar ist jede Lösung von S' auch Modell von S, denn: $x = \bigsqcup_{1 \leq i \leq k} e_j \implies x \sqsupseteq e_i \; \forall 1 \leq i \leq k$
Zu zeigen: Die kleinste Lösung $\bar{\eta}$ von S' ist sogar kleinstes Modell von S.
Dazu definieren wir zu S' die Funktion $F : (Vars \to D) \longrightarrow (Vars \to D)$ durch $(F(\eta))(x) = [f(x)](\eta)$
Wir hatten in Abschnitt 2.3 gesehen, daß F stetig und $\bar{\eta}$ der kleinste Fixpunkt von F ist.
Sei nun η irgendein Modell von S. Dann gilt für jedes x:
$\eta(x) \sqsupseteq [exp](\eta)$ sofern $x \sqsupseteq exp$ Ungleichung in S ist.
$\implies \eta(x) \sqsupseteq [f(x)](\eta)$ für alle $x \implies \eta \sqsupseteq F(\eta)$
Nach dem Satz von Knaster–Tarski ist aber $\bar{\eta} = \bigsqcap\{\eta \mid \eta \sqsupseteq F(\eta)\} \implies \bar{\eta} \sqsubseteq \eta$ □

Folgerung:
Um ein kleinstes Modell von S zu berechnen, genügt es, erst S' zu konstruieren, und dazu die kleinste Lösung zu berechnen.

3.4.2 Algorithmus zur Konstruktion des UGL S_E

Idee zur Lösung von Problem 2.:

- Versuche den expliziten Aufbau von S (bzw. S') zu vermeiden !
- Behandle Ausdrücke der Art $(e_1\ e_2)$ oder (`case` e_1 `of` *altlist*) nur dann, wenn sich der Wert für e_1 ändert !
- Verwalte die gefundenen Ungleichungen, deren 'Effekte' noch zu propagieren sind, in einer Worklist !

Algorithmus:

```
init();    /* initialisiert Resultatmenge R, Worklist W und erledigt leichte Fälle */
while (W != ∅) {
  [e]# ⊇ {d} := extract(W);
  propagate([e]# ⊇ {d});    /* propagiert entlang Variableninklusionen */
  switch (father_of(e)) {
    case (e1 e2):
      if (λx.e' == {d}) {
        add([(e1 e2)]# ⊇ [e']#);
        add([x]# ⊇ [e2]#);
        }
      break;
    case (case e of ···C x1 ... xn → e1 ···):
      if (C d1 ... dn == d) {
        add([case e of ...]# ⊇ [e1]#);
        add([x1]# ⊇ [d1]#);
          ⋮
        add([xn]# ⊇ [dn]#);
        }
      break;
    default: ;
    }   /* switch */
  }   /* while */
```

father_of(e) bezeichnet den Ausdruck, der e umfaßt. Die Funktionen **init**, **add** und **propagate** sind definiert als:

```
void init() {
  R := W := {[e]# ⊇ {e} | e ∈ E_pr}
  forall (e ∈ E_pr) {
    switch (e) {
      case (let x = e1 in e0)):
        add([x]# ⊇ [e1]#);
        add([e]# ⊇ [e0]#);
        break;
      case (μx.e'):
        add([x]# ⊇ [e]#);
        add([e]# ⊇ [e']#);
        break;
      case (case e of ···x → e1 ···):
        add([e]# ⊇ [e1]#);
        add([x]# ⊇ [e0]#);
        break;
      default: ;
      }   /* switch */
    }   /* forall */
}
```

```
void add([e]# ⊇ [e']#) {
  R := R ∪ {[e]# ⊇ [e']#}
  forall ([e']# ⊇ {d} ∈ R) {
    if ([e]# ⊇ {d} ∉ R) {
      R := R ∪ {[e]# ⊇ {d}};
      W := W ∪̇ {[e]# ⊇ {d}};
      }
    }
}

void propagate([e]# ⊇ {d}) {
  forall ([e']# ⊇ [e]# ∈ R) {
    if ([e']# ⊇ {d} ∉ R) {    /* neue Ungleichung gefunden */
      R := R ∪ {[e']# ⊇ {d}};
      W := W ∪̇ {[e']# ⊇ {d}};
      }
    }
}
```

Komplexität des Algorithmus':

Sei $\bar{\eta}$ das kleinste Modell des Ungleichungssystems, $\bar{R} \subseteq (Vars \times (Vars \cup E_{pr}))$ die Ergebnisrelation des Algorithmus' und $d := \bigsqcup_{e \in E_{pr}} (\sharp\bar{\eta}([e]^{\#}))$.
Es gilt stets: $d \leq |pr|$ (aber i. a. ist $d \leq 5$)

- Platzverbrauch: $O(\sharp\bar{R}) \leq O(d \cdot |pr|)$
- Zeitaufwand:

Zeitaufwand:		Gesamtzeit aller **propagate**–Aufrufe:	$O(\sharp\bar{R})$
	+	Zeit die im `switch`–Statement verbracht wird (ohne die Zeit für **add**):	$O(d \cdot \lvert pr \rvert)$
	+	Anzahl der **add**–Aufrufe:	$O(\sharp\bar{R})$
	+	Laufzeit der **add**–Routine:	$O(d)$
	+	Aufwand für die **init**–Routine:	$O(\lvert pr \rvert)$
	=	Gesamtlaufzeit:	$O(d \cdot \sharp\bar{R}) \leq O(d^2 \cdot \lvert pr \rvert)$ (für kleines d linear !)

Beobachtung:

Wann immer der Algorithmus eine Ungleichung $[e]^{\#} \supseteq [e']^{\#}$ findet, dann gilt auch $[e]^{\#} \supseteq \{e'\}$ (und umgekehrt):
$[e]^{\#} \supseteq \{e'\} \implies [e]\bar{\eta} \supseteq [e']\bar{\eta}$ für die Lösung $\bar{\eta}$
$\Longrightarrow$ Eine Unterscheidung zwischen $\{e\}$ und $[e]^{\#}$ ist für die Implementierung unerheblich !
$\Longrightarrow$ Modifiziere den Algorithmus so, daß er nur die Relation $R' \subseteq E_{pr} \times E_{pr}$ berechnet!

Beispiel: `f` $\equiv$ `first` und `r` $\equiv$ `rest`

```
let f = λl1.case l1 of (x:xs)→x
  in (let r = λl2.case l2 of (y:ys)→ys
    in f (r (1:[])) )
```

Für alle Teilausdrücke e gilt $(\{e\} \subseteq [e]^{\#}$.
Die weiteren Ungleichungen $[e]^{\#} \supseteq [e']^{\#}$ gibt die folgende Tabelle an:

e	e'	Ungleichungen aus	erzeugt von
`let f ...`	`(let r = ...in ...)`	`case (let ...):`	**init()**
`f`	`λl1.case ...`		
`let r ...`	`f (r (1:[]))`	`case (let ...):`	
`r`	`λl2.case ...`		
`let f ...`	`f (r (1:[]))`		
`f (r (1:[]))`	`case l1 ...`	`case` $(e_1 e_2)$:	`while`-Schleife
`l1`	`r (1:[])`		
`r (1:[])`	`case l2 ...`	`case` $(e_1 \;\; e_2)$:	
`l2`	`1:[]`		
`let r ...`	`case l1 ...`	**propagate()**	
`let f ...`	`case l2 ...`		
`case l2 ...`	`ys`		
`y`	`1`	`case (case ...):`	
`ys`	`[]`		
`case l2 ...`	`[]`		
`r (1:[])`	`[]`	`case (case ...):`	
`l1`	`[]`		

3.4.3 Korrektheit des Algorithmus'

Problem 3.:
Korrektheit des Algorithmus (in seiner ursprünglichen Form) relativ zum kleinsten Model des Ungleichungssystems.

Beobachtung:
$S_1 \subseteq S_2$ für Ungleichungssysteme $S_1, S_2 \implies \eta_1 \sqsubseteq \eta_2$, wobei η_1, η_2 für kleinste Modelle η_i von S_i (eine Vergrößerung des Ungleichungssystems führt also zu einer Vergrößerung des kleinsten Modells).

Es gilt:

a) Ungleichungen, die mit **init** generiert wurden, sind auch in S selbst enthalten.

b) Sei R eine Menge von Ungleichungen und R_1 eine Menge weiterer Ungleichungen, die durch **add**$([e]^\# \supseteq [e']^\#)$ hinzugefügt werden. Dann hat $R \cup \{[e]^\# \supseteq [e']^\#\}$ das gleiche kleinste Modell, wie $R \cup R_1 \cup \{[e]^\# \supseteq [e']^\#\}$

c) Analog gilt für **propagate**:
Sei R eine Menge von Ungleichungen und $[e]^\# \supseteq \{d\} \in R$. Sei R_1 die Menge der von **propagate**$([e]^\# \supseteq \{d\})$ erzeugten Ungleichungen. Dann haben R und $R \cup R_1$ das gleiche kleinste Modell.

d) Seien R_i die Menge der Ungleichungen nach dem i–ten Durchlauf der `while`–Schleife und η_i das kleinste Modell von R_i.
Man zeigt:
   - Wird im i–ten Durchlauf eine Ungleichung $[e]^\# \supseteq [e']^\#$ hinzugefügt, dann gibt es eine Ungleichung $([e]^\# \supseteq \{b\}); [e']^\#$ für eine Bedingung $\{b\} == \{d\} \cap [e_0]^\#$ mit $[b](\eta_i) \neq \emptyset$, d. h. $d \in \eta_{n-1}([e_0]^\#)$
   - $\eta_i \sqsubseteq \bar{\eta}$ für alle i

Sei N die Anzahl der `while`–Schleifendurchläufe. Wir folgern:
η_N ist ein Modell für S. Da $\eta_n \sqsubseteq \bar{\eta}$ von S folgt, daß η_N auch kleinstes Modell für S ist.
Dabei gewinnen wir η_N aus R_N einfach durch: $\eta_N([e]^{\#}) = \{d \mid [e]^{\#} \supseteq \{d\} \in R_N\}$

3.4.4 Korrektheit der Kontrollflußanalyse

Bleibt noch zu untersuchen, inwiefern das kleinste Modell eines Ungleichungssystems S 'aussagekräftig' ist relativ zur operationellen Semantik. Die operationelle Semantik beschreibt zwar, wie sich Terme entwickeln, ignoriert aber, wie sich Terme aus Ausdrücken $e \in E_{pr}$ zusammensetzen.

Idee: Wir wählen eine 'implementierungsnähere' operationelle Semantik !

a) Diese Implementierung verzögert die Substitution. So verwaltet etwa auch ein Compiler Paare (e, β), wobei β die aktuelle Bindung der Variablen in e beschreibt.

$$\beta \;::=\; \emptyset \mid \langle x \mapsto (e, \beta_1)\rangle + \beta_2$$

Dabei schreiben wir für $\langle x_1 \mapsto (e_1, \beta_1)\rangle + \langle x_2 \mapsto (e_2, \beta_2)\rangle + \cdots + \langle x_n \mapsto (e_n, \beta_n)\rangle$ kurz $\langle x_1 \mapsto (e_1, \beta_1), \ldots, x_n \mapsto (e_n, \beta_n)\rangle$.

b) Teilausdrücke werden auf einem Stack ausgewertet. Der momentan 'aktive' Teilausdruck steht oben auf dem Stack; Teilausdrücke, deren Auswertung suspendiert wurde, stehen darunter.

$$\begin{array}{l} \gamma \;::= (e, \beta)\gamma' \\ \gamma' ::= \varepsilon \mid ((.e), \beta)\gamma' \mid ((\texttt{case . of } p \to e; \ldots), \beta)\gamma' \end{array}$$

'.' bezeichnet den 'Rücksprungpunkt'.

Auf der Menge der Stacks definieren wir eine Relation $\to$ durch:

(1) $(x, \beta)\gamma' \to (\beta(x))\gamma'$

(2) $(\mu x.e, \beta)\gamma' \to (e, \langle x \mapsto (\mu x.e, \beta)\rangle + \beta)\gamma'$

(3) $(\texttt{let } x \;=\; e_1 \texttt{ in } e_0, \beta)\gamma' \to (e_0, \langle x \mapsto (e_1, \beta)\rangle + \beta)\gamma'$

(4) $(e_1\ e_2, \beta)\gamma' \to (e_1, \beta)(.e_2, \beta)\gamma'$

(5) $(\lambda x.e, \beta_1)(.e_2, \beta)\gamma' \to (e, \langle x \mapsto (e_2, \beta)\rangle + \beta_1)\gamma'$

(6) $((\texttt{case } e_0 \texttt{ of } x \to e_1; \mathit{altlist}), \beta)\gamma' \to (e_1, \langle x \mapsto (e_0, \beta)\rangle + \beta)\gamma'$

(7) $((\texttt{case } e_0 \texttt{ of } p \to e_1; \mathit{altlist}), \beta)\gamma' \to (e_0, \beta)((\texttt{case . of } p \to e_1; \mathit{altlist}), \beta)\gamma'$

(8) $(C\ d_1, \ldots, d_n, \beta_1)(\texttt{case . of } C\ x_1, \ldots, x_n \to e_1; \mathit{altlist}, \beta)\gamma'$
$\to (e_1, \beta')\gamma'$, wobei $\beta' = \langle x_1 \mapsto (d_1, \beta_1), \ldots, x_n \mapsto (d_n, \beta_1)\rangle + \beta$

(9) $(C\ d_1, \ldots, d_n, \beta_1)(\texttt{case . of } C'\ x_1, \ldots, x_n \to e_1; \mathit{altlist}, \beta)\gamma'$
$\to (C\ d_1, \ldots, d_n)(\texttt{case . of } \mathit{altlist}, \beta)\gamma'$,für $C \neq C'$

Beispiel: Append–Funktion `a`

$e \equiv$ `let a =` μ`a'.`λ`l.case l of []`$\rightarrow \lambda$`y.y;`
`(x:xs)`$\rightarrow \lambda$`y.(x:(a' xs y))`
`in a [1] []`

Wir starten die Berechnung mit (e,β) = (`let a =` μ`a'....in a [1] []`$,\emptyset$)
und erhalten die folgenden Stackbelegungen durch Anwendung der Reglen (i) der Relation $\rightarrow$:

$\rightarrow$ (`a [1] []`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$) (3)

$\rightarrow$ (`a [1]`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$) (4)
(`.[]`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$)

$\rightarrow$ (`a`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$) (4)
(`.[1]`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$)
(`.[]`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$)

$\rightarrow$ (μ`a'`.λ`l.case l of` $\ldots,\emptyset$) (1)
(`.[1]`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$)
(`.[]`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$)

$\rightarrow$ (λ`l.case l of` $\ldots$,$\langle$`a'` $\mapsto$ (μ`a'`.λ`l.case` $\ldots,\emptyset$)$\rangle$) (2)
(`.[1]`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$)
(`.[]`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$)

$\rightarrow$ (`case l of` $\ldots$, $\langle$`l` $\mapsto$ (`[1]`, $\langle$`a` $\mapsto$ (μ`a'`$\ldots,\emptyset$)$\rangle$)$\rangle$ + $\langle$`a'` $\mapsto$ (μ`a'`$\ldots,\emptyset$)$\rangle$) (5)
(`.[]`,$\langle$`a` $\mapsto$ (μ`a'`.λ`l.case l of`$\ldots,\emptyset$)$\rangle$)

$\rightarrow$ (`l`,$\langle$`l`$\mapsto$ (`[1]`,$\langle$`a`$\mapsto$ (μ`a'`$\ldots,\emptyset$)$\rangle$)$\rangle$ +$\langle$`a'`$\mapsto$ (μ`a'`$\ldots,\emptyset$)$\rangle$) (7)
(`case . of []`$\rightarrow$`[];`$\ldots$, $\langle$`l`$\mapsto$ (`[1]`,$\langle$`a`$\mapsto$ (μ`a'`$\ldots,\emptyset$)$\rangle$)$\rangle$ +$\langle$`a'`$\mapsto$ (μ`a'`$\ldots,\emptyset$)$\rangle$)
(`.[]`,$\langle$`a`$\mapsto$ (μ`a'.`λ `l.case l of` $\ldots,\emptyset$)$\rangle$)

$\rightarrow$ (`[1]`,$\langle$`a`$\mapsto$ (μ`a'.`λ `l.case l of`$\ldots,\emptyset$)$\rangle$) (1)
(`case . of []`$\rightarrow$`[];`$\ldots$, $\langle$`l`$\mapsto$ (`[1]`,$\langle$`a`$\mapsto$ (μ`a'`$\ldots,\emptyset$)$\rangle$)$\rangle$ +$\langle$`a'`$\mapsto$ (μ`a'`$\ldots,\emptyset$)$\rangle$)
(`.[]`,$\langle$`a`$\mapsto$ (μ`a'.`λ `l.case l of` $\ldots,\emptyset$)$\rangle$)

$\rightarrow$ (λ`y.(x:(a' xs y))`,$\langle$`x`$\mapsto$ (`1`,$\langle$`a`$\mapsto\ldots$$\rangle$), `xs`$\mapsto$ (`[]`,$\langle$`a`$\mapsto\ldots$$\rangle$)$\rangle$
+$\langle$`l`$\mapsto$ ($\ldots,\ldots$)$\rangle$ +$\langle$`a'`$\mapsto$ ($\ldots,\ldots$)$\rangle$) (9),(8)
(`.[]`,$\langle$`a`$\mapsto$ (μ`a'.`λ `l.case l of` $\ldots,\emptyset$)$\rangle$)

$\rightarrow$ (`x:(a' xs y)`,$\langle$`y`$\mapsto$ (`[]`,$\langle$`a`$\mapsto$ (μ`a'.`$\lambda\ldots,\emptyset$)$\rangle$)$\rangle$ +$\langle$`x`$\mapsto$ ($\ldots,\ldots$),`xs`$\mapsto$ ($\ldots,\ldots$)$\rangle$
+$\langle$`l`$\mapsto$ ($\ldots,\ldots$)$\rangle$+$\langle$`a'`$\mapsto$ ($\ldots,\ldots$)$\rangle$) (5)

Wir erhalten somit den partiellen Wert `x:(a' xs y)` mit der Bindung
$\langle$`y`$\mapsto$ (`[]`,$\ldots$), `x`$\mapsto$ (`1`,$\ldots$),`xs`$\mapsto$ (`[]`,$\ldots$), `l`$\mapsto$ (`[1]`,$\ldots$),`a'`$\mapsto$ (μ`a'`$\ldots,\ldots$)$\rangle$

Der Stack γ repräsentiert einen Ausdruck e, den man mittels einer Funktion u (*unroll*) berechnet.

u mit $u[\gamma] = e$ ist definiert durch:

$u[(e,\beta)] = e$, falls $Vars(e) \cap Dom(\beta) = \emptyset$
$u[(x,\beta)] = u[\beta(x)]$, falls $x \in Dom(\beta)$
$u[(e_1\ e_2),\beta] = u[(e_1,\beta)]\ u[(e_2,\beta)]$
$\vdots$
$u[(e_1,\beta_1)(.e_2,\beta_2)\gamma'] = u[(e'\ e_2,\beta_2)\gamma']$, wobei $e' = u[(e_1,\beta_1)]$
$u[(e_1,\beta_1)(\texttt{case . of}\ldots,\beta_2)\gamma'] = u[(\texttt{case}\ e'\ \texttt{of}\ldots,\beta_2)\gamma']$, wobei $e' = u[(e_1,\beta_1)]$

<u>Beispiel:</u>

Die Fortführung des Beispiels von oben ergibt
$u[\gamma] = \texttt{1:(}u[(\texttt{a}'\ \texttt{xs}\ \texttt{y}, \langle \texttt{y} \mapsto \ldots, \texttt{x} \mapsto \ldots, \texttt{xs} \mapsto \ldots, \texttt{l} \mapsto \ldots, \texttt{a}' \mapsto \ldots \rangle)]\texttt{)}$
$\quad = \texttt{1:((}\mu\texttt{a'.}\lambda\texttt{l...) [] [])}$

Es gilt folgender

<u>Satz:</u>

1) Aus $\gamma_1 \to \gamma_2$ folgt $u[\gamma_1] = u[\gamma_2]$ oder $u[\gamma_1] \to u[\gamma_2]$.
   ($\to$ bezeichnet hier die Auswertungsrelation auf Stacks)

2) Aus $e_1 \to e_2$ und $u[\gamma_1] = e_1$ folgt $\gamma_1 \overset{*}{\to} \gamma_2$ mit $u[\gamma_2] = e_2$.
   ($\to$ bezeichnet hier die Auswertungsrelation auf Termen)

D. h., daß die Berechnung auf Stacks der Berechnung auf Termen entspricht.

Die operationelle Semantik mit Stacks eignet sich, um die Korrektheit der Kontrollflußanalyse zu zeigen.
Dazu führen wir eine Abbildung $\alpha : \beta \mapsto M_{\supseteq}$ auf die Menge der Ungleichungen $M_{\supseteq}$ des Ungleichungssystems S_E ein (man bezeichnet $M_{\supseteq}$ auch als *set of constraints*). Diese Abbildung ist wie folgt definiert:

- $\alpha(\emptyset) = \emptyset$
- $\alpha(\langle x \mapsto (e, \beta)\rangle + \beta') = \{[x]^{\#} \supseteq [e]^{\#}\} \cup \alpha(\beta) \cup \alpha(\beta')$

α läßt sich fortsetzen zu Abbildungen $\alpha : \gamma \mapsto M_{\supseteq}$ durch:

- $\alpha(\varepsilon) = \emptyset$
- $\alpha((e, \beta)\gamma') = \alpha(\beta) \cup \alpha(\gamma')$

Sei S_E das Ungleichungssystem zu *pr*, $\overline{\eta}$ das kleinste Modell von S_E und R die Relation mit $R = \{[e]^{\#} \supseteq \{d\} \mid \overline{\eta}([e]^{\#}) \ni d\} \cup \{[e]^{\#} \supseteq [d]^{\#} \mid \overline{\eta}([e]^{\#}) \ni d\}$.
Dann haben R und S_E offenbar das gleiche kleinste Modell, nämlich $\overline{\eta}$.

<u>Satz:</u>

1) $\alpha(\emptyset) \subseteq R$
2) Gilt $\alpha(\beta) \subseteq R$ und $(e, \beta) \to \gamma$, dann ist $\alpha(\gamma) \subseteq R$.
3) Gilt $\alpha(\beta) \subseteq R$ und $(e, \beta) \overset{*}{\to} (e', \beta)$, dann ist $[e]^{\#} \supseteq [e']^{\#} \in R$.

<u>Beweisidee:</u> Induktion nach der Länge der Berechnung, wobei in jedem Schritt eine Fallunterscheidung entsprechend der Definition der Relation $\to$ vorzunehmen ist. $\square$

3.5 Notwendigkeitsanalyse

Bei der Notwendigkeitsanalyse stellt sich die Frage, 'wieviel' man von x auswerten muß, um eine Funktion $f(x)$ zu berechnen.

Beispiel:

Funktion	nötig vom Argument
$f(x) = 1$	nichts
$f(\mathit{True}) = \mathit{False}$ $f(\mathit{False}) = \mathit{True}$	alles
`rev l`	alles
`first l`	nur Top–Konstruktor und 1. Element von `l`

Wissen wir, daß das Argument 'ganz' benötigt wird, kann dieses vorneweg ausgewertet und das Resultat 'by value' übergeben werden. Dies führt zu effizienteren Programmen.
Eine andere Anwendung finden wir in der automatischen Parallelisierung:
Benötigt der Funktionsaufruf $f\ e_1\ e_2$ mit Sicherheit e_1 und e_2 'ganz', so kann man zwei unabhängige parallele Prozesse zur Auswertung von e_1 bzw. e_2 starten.

Problem:
Eine Anforderung an das Ergebnis wird transformiert in eine Anforderung an die Argumente. Wir haben somit eine Rückwärtsanalyse vor uns.

Wir stellen erst einige **Vereinfachungen** an, indem wir auf funktionale Argumente, funktionale Resultate (wir betrachten nur Werte) und auf lokale Funktionsdefinitionen im Hauptausdruck verzichten.

Wir untersuchen also ein System rekursiver Funktionsdefinitionen mit Patternmatching, allerdings jetzt auf mehreren Argumenten mit beliebigen Pattern (aber immer noch ohne Variablenwiederholung).
Außerdem gibt es nur einen Datentyp.
D. h. wir erhalten folgende Grammatik:

$$\begin{array}{lcl}
pr & ::= & tdef;\ fdefs \\
tdef & ::= & t\ ==\ cdefs \\
cdefs & ::= & \varepsilon \mid C\ t \ldots t\ +\ cdefs \\
e & ::= & x \mid C \mid (C\ e_1 \ldots e_k) \mid (f\ e_1 \ldots e_k) \\
fdefs & ::= & \varepsilon \mid f\ p_1 \ldots p_n\ =\ e;\ fdefs \\
p & ::= & C \mid x \mid (C\ p_1 \ldots p_n)
\end{array}$$

Beispiel:

`app [] y = y; app (x:xs) y = x:(app xs y)` ist ein Programm in diesem Sinne.

Sei $\mathcal{F}$ die Menge der definierten Funktionen des Programms.

3.5.1 Der Mengenansatz

Vorgehensweise:

- Ordne einer n–stelligen Funktion f eine Menge $[f]^\# \subseteq \{1, \ldots, n\}$ zu, wobei $j \in [f]^\#$ bedeutet: f benötigt das 'ganze' j–te Argument (sofern der Wert von $(f\ \ldots)$ 'ganz' benötigt wird).

- Stelle ein Gleichungssystem S_0 auf mit $[f]^\# = F_f,\ f \in \mathcal{F}$

Umsetzung

1. Sei f definiert durch die Alternativen $alt_1; \ldots; alt_k$. Benötige die Alternative alt_j sicher alle Argumente aus $[alt_j]^\#$ für $1 \leq j \leq k$, dann setzen wir:
   $[f]^\# = [alt_1]^\# \cap \cdots \cap [alt_k]^\#$ (die Operation $\cap$ garantiert uns sichere Informationen).

2. Sei $alt \equiv f\ p_1 \ldots p_k = e$ eine Alternative für f und
   $$Vars := \bigcup_{j=1}^{k} Vars(p_j),$$
   wobei $Vars(p_j)$ die Menge der im Pattern p_j enthaltenen Variablen ist.
   Sei $[e]^\# \subseteq Vars$ eine Menge von Variablen, deren Werte mit Sicherheit 'ganz' benötigt wird, wenn e 'ganz' ausgewertet werden soll. Aus $[e]^\#$ wollen wir auf $[alt]^\#$ zurückschließen mittels einer Transformation $[\![p_1, \ldots, p_n]\!]^\# : \mathcal{P}(Vars) \longrightarrow \mathcal{P}(\{1, \ldots, n\})$, die definiert ist durch:
   $$[\![p_1, \ldots, p_n]\!]^\#(A) := \{i \mid Vars(p_i) \subseteq A\}.$$
   Dann setzen wir $[alt]^\# := [\![p_1, \ldots, p_n]\!]^\#([e]^\#)$.

3. $[e]^\# \subseteq Vars$ definieren wir induktiv über der Struktur von e:
   $$[C]^\# = \emptyset$$
   $$[x]^\# = \{x\}$$
   $$[C\ e_1 \ldots e_k]^\# = [e_1]^\# \cup \cdots \cup [e_k]^\#$$
   $$[f\ e_1 \ldots e_n]^\# = [f]^\# * ([e_1]^\#, \ldots, [e_n]^\#),$$
   $$\text{wobei } A * (B_1, \ldots, B_n) := \bigcup_{j \in A} B_j$$

Beispiel: $alt_1 \equiv$ `app [] y = y` $alt_2 \equiv$ `app (x:xs) y = x:(app xs y)`

$$[\texttt{app}]^\# = [alt_1]^\# \cap [alt_2]^\#$$
$$[alt_1]^\# = [\![\texttt{[]},\texttt{y}]\!]^\#(\{\texttt{y}\}) = \{1, 2\}$$
$$[alt_2]^\# = [\![\texttt{(x:xs)},\texttt{y}]\!]^\#([\texttt{x} : (\texttt{app xs y})]^\#)$$
$$[\texttt{x} : (\texttt{app xs y})]^\# = \{\texttt{x}\} \cup [\texttt{app xs y}]^\#$$
$$[\texttt{app xs y}]^\# = [\texttt{app}]^\# * (\{\texttt{xs}\}, \{\texttt{y}\})$$

Satz:

(a) Für jede Patternliste $\underline{p} = p_1, \ldots, p_n$ mit Variablenmenge $Vars$ ist $[\![\underline{p}]\!] : \mathcal{P}(Vars) \longrightarrow \mathcal{P}(\{1, \ldots, n\})$ monoton.

(b) Die Operation $* : \mathcal{P}(\{1, \ldots, n\}) \times (\mathcal{P}(Vars))^n \longrightarrow \mathcal{P}(Vars)$ ist ebenfalls monoton.

Wir schließen: Unser Gleichungssystem besitzt eine eindeutig bestimmte kleinste, bzw. größte Lösung.

Die größte Lösung eines Gleichungssystems bestimmt man, indem man die partielle Ordnung umkehrt, und die Iteration mit dem größten Element startet.

e	kleinste Lösung $[e]^{\#}$	größte Lösung $[e]^{\#}$	
alt_1	$\{1,2\}$	$\{1,2\}$	
`app xs y`	$\emptyset$	$\{\texttt{xs},\texttt{y}\}$	
`x:(app xs y)`	$\{\texttt{x}\}$	$\{\texttt{x},\texttt{xs},\texttt{y}\}$	
alt_2	$\emptyset$	$\{1,2\}$	
`app`	$\emptyset \leftarrow$	$\{1,2\} \leftarrow$	kleinste Fixpunkte

Die Suche nach der größten Lösung entspricht einer optimistischen Vorgehensweise. Die gefundene größte Lösung für `app` ($\{1,2\}$) bedeutet, daß `app` alle Argumente benötigt.

Beispiel: alt_1 $\equiv$ `if True x y = x` alt_2 $\equiv$ `if False x y = y`

$$[alt_1]^{\#} = [\![\texttt{True},\texttt{x},\texttt{y}]\!]^{\#}(\{\texttt{x}\}) = \{1,2\}$$
$$[alt_2]^{\#} = [\![\texttt{False},\texttt{x},\texttt{y}]\!]^{\#}(\{\texttt{y}\}) = \{1,3\}$$
$$\Longrightarrow [\texttt{if}]^{\#} = [alt_1]^{\#} \cap [alt_2]^{\#} = \{1\}$$

Problem: Betrachte $e \equiv$ `if` e_1 e_2 e_3 !
Wird x ganz benötigt, sofern e_2 ganz benötigt wird, und ganz benötigt, sofern e_3 ganz benötigt wird, dann wird x offenbar auch ganz für die Auswertung von e benötigt ! Diese Information liefert unsere Analyse bisher nicht !
$\Longrightarrow$ Wir müssen eine Verfeinerung vornehmen: Dazu fassen wir die Beiträge der Alternativen alt_j für f nicht sofort zu einem Wert $[f]^{\#} = [alt_1]^{\#} \cap \cdots \cap [alt_k]^{\#}$ zusammen, der dann auf $([e_1]^{\#}, \ldots, [e_k]^{\#})$ angewendet wird, sondern wenden **erst** die Alternativen an, und bilden nachher den Durchschnitt über die so entstandene Menge !

$$\text{D. h.: } [f\ e_1 \ldots e_n]^{\#} = \bigcap_{j=1}^{n} \Big([alt_j]^{\#} * ([e_1]^{\#}, \ldots, [e_n]^{\#}) \Big)$$

Für das Beispiel mit `if` erhalten wir:
$[\texttt{if}\ e_1\ e_2\ e_3]^{\#} = ([e_1]^{\#} \cup [e_2]^{\#}) \cap ([e_1]^{\#} \cup [e_3]^{\#}) = [e_1]^{\#} \cup ([e_2]^{\#} \cap [e_3]^{\#})$

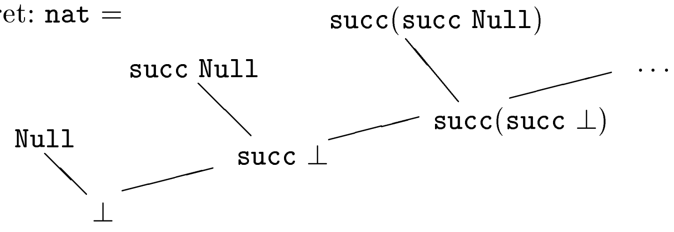
Frage: Ist diese Vorgehensweise korrekt ?
Bevor wir Werte rückwärts durch Funktionen propagieren, wollen wir erst einmal Werte vorwärts propagieren.

3.5.2 Denotationelle Semantik für rekursive Funktionsdefinitionen

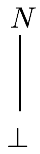
Sei D ein semantischer Bereich für (konkrete oder abstrakte) Werte. D sei eine partielle Ordnung mit kleinstem Element $\perp$.

Beispiel:

- konkret: `nat` =



- abstrakt: *two* =



$N \;\hat{=}$ Wert enthält evtl. kein $\perp$

$\perp \;\hat{=}$ Wert enthält sicher $\perp$

Allgemeine Vorgehensweise:
Einem n–stelligen Funktionssymbol f ordnen wir eine n–stellige Funktion $[f]$ mit Definitionsbereich D^n zu. Zur Berechnung (bzw. Definition) von f stellen wir ein Gleichungssystem $[f] = F_f,\ f \in \mathcal{F}$ auf !
Die **kleinste** Lösung dieses Gleichungssystems heißt *denotationelle Semantik*.

Sei f definiert durch die Alternativen $alt_1; \ldots ; alt_k$. Wir würden gerne $[f]$ aus den Funktionen $[alt_1], \ldots, [alt_k]$ konstruieren: $[f] = [alt_1] + \cdots + [alt_k]$
Nun stellt sich die Frage was der Operator '+' bewirkt. Wir wählen einfach $+ \equiv \sqcup$.

Probleme:

- $\sqcup$ führt alle Alternativen 'parallel' aus und liefert die kleinste obere Schranke der Ergebnisse. Das ist korrekt, sofern jeweils höchstens eine Alternative zutrifft. Andernfalls erhalten wir evtl. ein 'größeres' Ergebnis (ist aber ok).
- $\sqcup$ existiert evtl. gar nicht !

Ausweg: Wir konstruieren zur partiellen Ordnung D die Menge $P(D)$ wie folgt:

Definition:
Sei D eine partielle Ordnung. Dann ist der HOARE*'sche Powerdomain* $P(D)$ definiert als $P(D) := \{\emptyset \neq A \subseteq D \mid x \in A \implies x{\downarrow} \subseteq A\}$, wobei $x{\downarrow} := \{y \in D \mid y \sqsubseteq x\}$.

Charakterisierungssatz:
Sei D eine partielle Ordnung.

(1) $P(D)$ ist ein vollständiger Verband mit $\subseteq$ als Ordnung und $\cup$ als kleinster oberer Schranke.

(2) Für partielle Ordnungen D_1 und D_2 ist $P(D_1 \times D_2) \hookleftarrow P(D_1) \times P(D_2)$.
(d. h. es gibt eine injektive und stetige Abbildung von $P(D_1) \times P(D_2)$ nach $P(D_1 \times D_2)$)

(3) Für einen vollständigen Verband $\overline{D}$ läßt sich jede monotone Funktion $f : D \longrightarrow \overline{D}$ (eindeutig) fortsetzen zu einer Abbildung $P(f) : P(D) \longrightarrow \overline{D}$, die definiert ist durch

$$(P(f))(A) := \bigsqcup_{a \in A} f(a).$$

(a) $P(f)$ ist additiv, d. h. $P(f)$ kommutiert mit allen **nichtleeren** Vereinigungen:

$$(P(f))(\bigcup_{i\in I} A_i) = \bigsqcup_{i\in I}(P(f))(A_i) \qquad I \neq \emptyset,$$

wobei I eine beliebige Indexmenge ist.

(b) Sei $s : D \longrightarrow P(D)$ die Einbettung $s(x) = x{\downarrow}$. Dann ist s monoton, $P(s) = id_{P(D)}$ und $f = P(f) \circ s$.
Im kommutativen Diagramm:

$$\begin{tikzcd}
D \arrow[r, "f"] \arrow[d, "s"'] \arrow[dr, phantom, "\circlearrowright"] & \overline{D} \\
P(D) \arrow[ur, "P(f)"'] &
\end{tikzcd}$$

(c) Seien $f : D_2 \longrightarrow \overline{D}$, $g : D_1 \longrightarrow P(D_2)$ monotone Abbildungen und $\overline{D}$ ein vollständiger Verband. Dann gilt: $P(P(f) \circ g) = P(f) \circ P(g)$.
Im Diagramm:

$$\begin{tikzcd}
D_1 \arrow[dr, "g"'] & & D_2 \arrow[dr, "f"] & \\
& P(D_2) \arrow[rr, "P(f)"] & & \overline{D} \\
P(D_1) \arrow[ur, "P(g)"] \arrow[urrr, "P(P(f)\circ g)"'] & & &
\end{tikzcd}$$

Insbesondere kann man zwischen f und g eine Verknüpfung $\diamond$ definieren durch
$$f \diamond g = P(f) \circ g.$$
Die Verknüpfung $\diamond$ ist assoziativ mit neutralem Element s.

<u>Beweis:</u>

- zu 3(b):

$$\begin{aligned}
(P(s))(A) &= \bigsqcup_{a\in A} s(a) = \bigcup_{a\in A} a{\downarrow} = A \\
(P(f)\circ s)(a) &= (P(f))(s(a)) = (P(f))(a{\downarrow}) = \bigsqcup_{d\in a\downarrow} f(d) \\
&= \bigsqcup_{d\sqsubseteq a} f(d) = f(a)
\end{aligned}$$

- zu 3(c):

$$\begin{aligned}
(P(P(f)\circ g))(A) &= \bigsqcup_{a\in A}(P(f)\circ g)(a) = \bigsqcup_{a\in A}(P(f))(g(a)) \\
&= (P(f))(\bigcup_{a\in A} g(a)), \text{ da } P(f) \text{ additiv} \\
&= (P(f))((P(g))(A)) = (P(f)\circ P(g))(A)
\end{aligned}$$

<u>Beispiel:</u>

$$D = two = \begin{tikzcd} N \arrow[d, dash] \\ \perp \end{tikzcd} \quad \Rightarrow \quad P(two) = \begin{tikzcd} \{N, \perp\} \arrow[d, dash] \\ \{\perp\} \end{tikzcd}$$

$$B = \begin{tikzcd}[column sep=small] \texttt{True} \arrow[dr, dash] & & \texttt{False} \arrow[dl, dash] \\ & \perp & \end{tikzcd} \quad \Rightarrow \quad P(B) = \begin{tikzcd}[column sep=small] & \{\texttt{True}, \texttt{False}, \perp\} \arrow[dl, dash] \arrow[dr, dash] & \\ \{\texttt{True}, \perp\} \arrow[dr, dash] & & \{\texttt{False}, \perp\} \arrow[dl, dash] \\ & \{\perp\} & \end{tikzcd}$$

$D = \texttt{nat} \implies P(\texttt{nat})$ enthält beispielsweise die Elemente $\{\perp\}, \{\texttt{Null}, \perp\}$,
$\{\texttt{Null}, \texttt{succ Null}, \texttt{succ } \perp, \perp\}, \{\texttt{succ(succ } \perp), \texttt{succ } \perp, \perp\}, \ldots$

Den Wert einer n–stelligen Funktion f geben wir als monotone Funktion $[f]\colon D^n \longrightarrow P(D)$ an. Wird f durch die Alternativen $alt_1, \ldots alt_k$ definiert, dann gilt:

$[f] = [alt_1] \sqcup \cdots \sqcup [alt_k]$, mit $[f](\underline{d}) = [alt_1](\underline{d}) \cup \cdots \cup [alt_k](\underline{d})$,

wobei $\underline{d}$ ein Argumenttupel $\underline{d} = (d_1, \ldots d_n) \in D^n$ ist.

Nun müssen wir noch für eine Funktionsalternative $alt \equiv f\ p_1 \ldots p_n = e$ eine monotone Funktion $[alt]\colon D^n \longrightarrow P(D)$ definieren. Wir gehen dabei in 3 Schritten vor:

1. Für $\underline{p} = p_1, \ldots, p_n$ sei *Vars* wiederum die Menge der Variablen der Patternliste $\underline{p}$.
   Sei im weiteren $P(D)_\emptyset := P(D) \cup \{\emptyset\}$, geordnet durch die Mengeninklusion (wir brauchen nämlich die leere Menge fr das Patternmatching).
   Wir benötigen eine monotone Funktion $[\![\underline{p}]\!]\colon D^n \longrightarrow P(\mathit{Vars} \to D)_\emptyset$.

   Für das Beispiel $D = \texttt{nat}$ ist $[\![\underline{p}]\!]$ ist definiert durch:

   $$[\![\underline{p}]\!](d_1, \ldots, d_n) := \begin{cases} \rho\!\downarrow, & \text{falls } p_j\,\rho = d_j \quad \forall\, 1 \le j \le n \\ \emptyset, & \text{sonst (d. h. Patternmatching ist fehlgeschlagen)} \end{cases}$$

   ρ ist dabei die Substitution (Abbildung), in der alle Vorkommen von Variablen $x \in \mathit{Vars}(p_j)$ durch Werte $\rho(x)$ ersetzt werden und $\perp$ die Abbildung ρ ist mit $\rho(x) = \perp\ \forall x \in \mathit{Vars}$.

   Man erhält beispielsweise:

   - $[\![\texttt{Null}, \texttt{Succ(Succ y)}, \texttt{Succ z}]\!](\texttt{Null}, \texttt{Succ(Succ Null)}, \texttt{Succ(Succ Null)})$
     $$= \rho\!\downarrow \text{ mit } \rho = \begin{cases} \texttt{y} \mapsto \texttt{Null} \\ \texttt{z} \mapsto \texttt{Succ Null} \end{cases}$$

     Fassen wir ρ wieder als Substitution von Variablen durch Elemente aus `nat` auf, dann schreiben wir auch:
     $\rho\!\downarrow = \{[\texttt{Null/y}, \texttt{Succ Null/z}], [\perp\texttt{/y}, \texttt{Succ Null/z}], [\perp\texttt{/y}, \texttt{Succ } \perp\texttt{/z}], \ldots, [\perp\texttt{/y}, \perp\texttt{/z}]\}$
   - $[\![\texttt{Null}, \texttt{Succ(Succ y)}, \texttt{Succ z}]\!](\texttt{Succ } \perp, \texttt{Succ(Succ Null)}, \perp) = \emptyset$

   Für das Beispiel $D = two$ liefert $[\![\underline{p}]\!](d_1, \ldots, d_n)$ die Menge aller ρ mit:
   $d_j = \perp \implies \rho(x) = \perp$ für ein $x \in \mathit{Vars}(p_j)$.

2. Nehmen wir an, wir hätten eine monotone Funktion $[e] : (\mathit{Vars} \to D) \longrightarrow P(D)$ gefunden. Dann setzen wir $[alt](\underline{d}) := \bigsqcup_{\rho \in [\underline{p}](\underline{d})} [e](\rho)$.
   D. h. $[alt] = [e] \odot [\![\underline{p}]\!]$, wobei $\odot$ definiert ist durch: $\varphi \odot [\![\underline{p}]\!] := \bigsqcup_{\rho \in [\underline{p}](\underline{d})} \varphi(\rho)$.
   Wir setzen $\bigsqcup_{\rho \in \emptyset} \varphi(\rho) := \bot\!\downarrow$.

3. Nun gilt es noch die monotone Abbildung $[e] : (\mathit{Vars} \to D) \longrightarrow P(D)$ induktiv über die Struktur von $[e]$ zu konstruieren (1. Repräsentation):
   $[C] : (\mathit{Vars} \to D) \longrightarrow P(D)$ sei für 0–stellige Konstruktoren geeignet zu D definiert
   $[x](\rho) = (\rho(x))\!\downarrow$
   $[C\, e_1 \ldots e_n](\rho) = P[C](\overbrace{[e_1](\rho) \times \cdots \times [e_n](\rho)}^{\in P(D)^n \hookrightarrow P(D^n)})$, für geeignet definierte Funktionen $[C] : D^n \longrightarrow P(D)$ mit $n \geq 1$
   $[f\ e_1 \ldots e_n](\rho) = P[f]([e_1](\rho) \times \cdots \times [e_n](\rho))$

   Beispiele für die Definition von $[C]$:

$$D = \texttt{nat} : [\texttt{Null}](\rho) = \{\texttt{Null}, \bot\} = \texttt{Null}\!\downarrow$$
$$[\texttt{Succ}](d) = (\texttt{Succ}\ d)\!\downarrow$$
$$D = \mathit{two} : [\texttt{Null}](\rho) = \{N, \bot\} = N\!\downarrow$$
$$[\texttt{Succ}](d) = \begin{cases} \bot\!\downarrow, & \text{falls } d = \bot \\ N\!\downarrow, & \text{falls } d = N \end{cases} \qquad \text{d. h. } [\texttt{Succ}] = s$$

<u>Eine kompaktere Darstellung für 3. (2. Repräsentation):</u>

$[x] = s \circ \pi_x$, wobei π_x die Projektion in Richtung x ist.
$[C\ e_1 \ldots e_n] = [C] \diamond ([e_1] \times \cdots \times [e_n]) \circ \Delta_n$, wobei $\Delta_n : D^n \longrightarrow D^{n \times n}$ definiert ist durch $\Delta_n(\underline{d}) = (\underline{d}, \ldots, \underline{d})$.
$[f\ e_1 \ldots e_n] = [f] \diamond ([e_1] \times \cdots \times [e_n]) \circ \Delta_n$

<u>Fazit:</u> Wir erhalten ein Gleichungssystem für $[f]$, $f \in \mathcal{F}$ (unter Zuhilfenahme von Hilfsvariablen) mit Operatoren $\circ, \diamond, \odot, \times, \sqcup$ und 'Konstanten' $[C], \Delta_n, [\![\underline{p}]\!], s \circ \pi_x$.
Auch hier wollen wir wieder nach der kleinsten Lösung des Gleichungssystems suchen.

<u>Beispiel:</u> Gleichungssystem zu `add`

$alt_1 \equiv$ `add Null y = y` $\qquad alt_2 \equiv$ `add (Succ x) y = Succ (add x y)`

$$[\texttt{add}] = [alt_1] \sqcup [alt_2]$$
$$[alt_1] = (s \circ \pi_{\texttt{y}}) \odot [\![\texttt{Null}, \texttt{y}]\!]$$
$$[alt_2] = [\texttt{Succ (add x y)}] \odot [\![\texttt{Succ x}, \texttt{y}]\!]$$
$$[\texttt{Succ (add x y)}] = [\texttt{Succ}] \diamond [\texttt{add x y}]$$
$$[\texttt{add x y}] = [\texttt{add}] \diamond ((s \circ \pi_{\texttt{x}}) \times (s \circ \pi_{\texttt{y}})) \circ \Delta_2$$

<u>Satz:</u>

1. Die Operatoren $\circ, \diamond, \odot, \times, \sqcup$ sind stetig.

2. Das oben konstruierte Gleichungsystem besitzt eine kleinste Lösung η mit:
   $\eta([f]) \in [D^n \to P(D)]_m \quad$ (f n–stellige Funktion).
   Dabei ist η gegeben durch:

$$\eta = \bigsqcup_{j \geq 0} \eta^{(j)} \text{ mit } \eta^{(0)}([f]) = \bot \text{ und } \eta^{(j)}([f]) = F_f(\eta^{(j-1)}) \text{ für } j > 0$$

(dabei ist $\bot$ diejenige Funktion in $[D^n \to P(D)]_m$ mit $\underline{d} \mapsto \{\bot\} \;\; \forall\, \underline{d} \in D^n$).

3. Seien $w_1, \ldots, w_n$ Werte und '$\Rightarrow$' die Auswertungsrelation für Terme.
$f\ w_1 \ldots w_n \ \stackrel{*}{\Rightarrow}\ w$ impliziert dann $w \in [f](w_1, \ldots, w_n)$, sofern wir für D die partielle Ordnung $\mathcal{T}$ der (partiellen) Werte wählen.
$\mathcal{T}$ ist gegeben durch:
   $\bot \in \mathcal{T}$
   $C \in \mathcal{T}$ für 0–stellige Konstruktoren C
   $(C\ t_1 \ldots t_n) \in \mathcal{T}$ für n–stellige Konstruktoren C, sofern $t_1, \ldots, t_n \in \mathcal{T}$
Die Ordnung auf $\mathcal{T}$ ist gegeben durch:
   $\bot \sqsubseteq t \;\; \forall\, t \in \mathcal{T}$
   $t_i \sqsubseteq t_i' \;\; \forall\, 1 \leq i \leq n \implies (C\ t_1 \ldots t_n) \sqsubseteq (C\ t_1' \ldots t_n')$

Beweis:
1. $\surd$
2. Folgerung aus 1.
3. mittels Induktion über die Länge einer Berechnungsfolge $f\ w_1 \ldots w_n \ \Rightarrow \cdots \Rightarrow\ w$ □

Achtung: Eine effiziente Lösung des Gleichungssystems ist i. a. nicht möglich, da $\mathcal{T}$, und damit erst recht $[\mathcal{T}^n \to \mathcal{T}]_m$, nicht die Ascending–Chain–Condition erfüllt !
Aber: Die Lösung über *two* ist berechenbar, da dieser Bereich endlich ist.

Fragen:

1. Wie effizient kann eine solche Lösung über *two* berechnet werden ?
2. In welchem Zusammenhang steht eine Lösung über *two* mit der über $\mathcal{T}$?
3. Was hat dies mit der Notwendigkeitsanalyse zu tun ?

3.5.3 Berechnung der denotationellen Semantik über *two*

Problem: Die Bereiche für Variablen des Gleichungssystems sind Mengen von Funktionen.

1. Ansatz:
Repräsentiere jede Funktion $[f], [alt], [e]$ als Wertetabelle, und berechne dann sukzessive die Tabelleneinträge!
Statt $\mathcal{T}$ verwenden wir nun den Bereich *two*. Wir definieren 'abstrakte Konstruktoranwendungen' und 'abstraktes Patternmatching':

$[C](\rho) = N\!\downarrow$, falls C ein 0–stelliger Konstruktor ist.
$[C](d_1 \ldots d_n) = N\!\downarrow \;:\Leftrightarrow\; \forall\, 1 \leq i \leq n : d_i = N$, falls C ein n–stelliger Konstruktor mit $n > 0$ ist.
$[\![\underline{p}]\!] : two^n \longrightarrow P(Vars \to two)_\emptyset$

Beispiel: Berechnung von [add] über $D{=}two$ (mittels 1. Repräsentation)

$alt_1 \equiv$ `add Null y = y` $\quad alt_2 \equiv$ `add (Succ x) y = Succ (add x y)`

$$[\texttt{add}] = \bigsqcup_{j\geq 0} [\texttt{add}]^{(j)} \qquad [\texttt{add}]^{(j)} = F_{\texttt{add}}([\texttt{add}] \mapsto [\texttt{add}]^{(j-1)}),\ j > 0$$

$$[\texttt{add}]^{(0)}(d_1, d_2) = \bot \quad \forall\, d_1, d_2 \in two$$

$$[\texttt{add}]^{(1)}(\bot,\bot) = [alt_1]^{(1)}(\bot,\bot) \cup [alt_2]^{(1)}(\bot,\bot)$$

$$[alt_1]^{(1)}(\bot,\bot) = ([\texttt{y}] \odot [\![\texttt{Null}, \texttt{y}]\!])(\bot,\bot) = \bigsqcup_{\rho\in\emptyset} [\texttt{y}](\rho) = \bot\!\downarrow$$

$$\begin{aligned} [alt_2]^{(1)}(\bot,\bot) &= ([\texttt{Succ (add x y)}]^{(1)} \odot [\![\texttt{Succ x}, \texttt{y}]\!])(\bot,\bot) \\ &= \bigsqcup_{\rho\in\{[\texttt{x}\mapsto\bot, \texttt{y}\mapsto\bot]\}} [\texttt{Succ (add x y)}]^{(1)}(\rho) \end{aligned}$$

$$[\texttt{Succ (add x y)}]^{(1)}(\rho) = (\underbrace{P[\texttt{Succ}]}_{=P(s)=id})([\texttt{add x y}]^{(1)}(\rho)) = [\texttt{add x y}]^{(1)}(\rho)$$

$$[\texttt{add x y}]^{(1)}(\rho) = (\underbrace{P[\texttt{add}]^{(0)}}_{=P(\bot)})((\rho(\texttt{x}))\!\downarrow \times (\rho(\texttt{y}))\!\downarrow) = \bot\!\downarrow$$

$$\Longrightarrow\ [\texttt{add}]^{(1)}(\bot,\bot) = [alt_1]^{(1)}(\bot,\bot) \cup [alt_2]^{(1)}(\bot,\bot) = \bot\!\downarrow \cup \bot\!\downarrow = \bot\!\downarrow$$

Mit ähnlicher Rechnung findet man:

$$\begin{aligned} [alt_1]^{(1)}(N,N) &= ([\texttt{y}] \odot [\![\texttt{Null}, \texttt{y}]\!])(N,N) \\ &= [\texttt{y}](\rho_1) \sqcup [\texttt{y}](\rho_2) \text{ mit } \rho_1 = [\texttt{y} \mapsto \bot],\ \rho_2 = [\texttt{y} \mapsto N] \\ &= (\rho_1(\texttt{y}))\!\downarrow \cup (\rho_2(\texttt{y}))\!\downarrow = \bot\!\downarrow \cup N\!\downarrow = N\!\downarrow \\ &\ \ \vdots \end{aligned}$$

$$\Longrightarrow\ [\texttt{add}]^{(1)}(N,N) = N\!\downarrow; \quad [\texttt{add}]^{(1)}(N,\bot) = [\texttt{add}]^{(1)}(\bot,N) = \bot\!\downarrow$$

Das liefert die Wertetabelle für $[\texttt{add}]^{(1)}$. Damit können wir dann die Wertetabelle für $[\texttt{add}]^{(2)}$ berechnen usw., bis sich diese nicht mehr ändert.

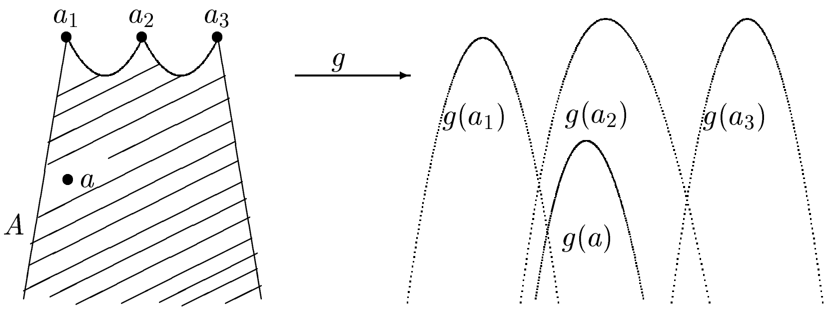
Verbesserung:

(1) *Beseitige das Rechnen mit mehrelementigen Mengen !*

Beobachtung: $P(two) = \{\bot\!\downarrow \sqsubset N\!\downarrow\}$

Wollen wir $(P(g))(A)$ für $A \in P(two^n)$ und $g : two^n \longrightarrow P(two)$ berechnen, dann genügt es offenbar, g auf die maximalen Elemente von A anzuwenden.

Im Bild:



D. h.: Sei $\{a_1, \ldots, a_k\}$ die Menge der maximalen Elemente von A. Dann ist

$$(P(g))(A) = \bigcup_{j=1}^{k} g(a_j).$$

Weiterhin liegt der Wert einer Funktionsanwendung in $P(two)$, und ist damit entweder $\perp\downarrow$ oder $N\downarrow$. Die einzige Ausnahme bildet $[\![\underline{p}]\!]$: Diese Abbildung liefert nämlich eine Menge von Variablenbelegungen, von der wir aber auch nur die maximalen Elemente betrachten müssen. Damit können wir das ursprüngliche Gleichungssystem vereinfachen:

$$\begin{aligned}
&[f](\underline{d}) = [alt_1](\underline{d}) \cup \cdots \cup [alt_k](\underline{d})\\
&[alt](\underline{d}) = \bigcup_{\substack{\rho \in [\underline{p}](\underline{d})\\ \rho \text{ maximal}}} [e](\rho)\\
&[C](\rho) = N\downarrow, \text{ falls } C \;\; 0\text{–stelliger Konstruktor}\\
&[x](\rho) = (\rho(x))\downarrow\\
&[C\ e_1 \ldots e_n](\rho) = [e_1](\rho) \cap \ldots \cap [e_n](\rho), \text{ falls } C\ n\text{–stelliger Konstruktor mit } n > 0\\
&[f\ e_1 \ldots e_n](\rho) = \bigcup_{\substack{\underline{d} \in [e_1](\rho) \times \cdots \times [e_n](\rho)\\ \underline{d} \text{ maximal}}} [f](\underline{d}) \overset{(\star)}{=} [f]([e_1](\rho), \ldots, [e_n](\rho))
\end{aligned}$$

($(\star)$ weil $[e_1](\rho) \times \cdots \times [e_n](\rho)$ einelementig ist)

Da die Bereiche two und $P(two)$ isomorph sind ($\{\perp\downarrow \sqsubset N\downarrow\} \cong \{\perp \sqsubset N\}$), ordnen wir n–stelligen Funktionen f Werte in $[two^n \to two]_m$ zu.
Wir erhalten als Gleichungssystem:

$$\begin{aligned}
&[f](\underline{d}) = [alt_1](\underline{d}) \sqcup \cdots \sqcup [alt_k](\underline{d})\\
&[alt](\underline{d}) = \bigsqcup_{\substack{\rho \in [\underline{p}](\underline{d})\\ \rho \text{ maximal}}} [e](\rho)\\
&[C](\rho) = N, \text{ falls } C \text{ 0–stelliger Konstruktor}\\
&[x](\rho) = \rho(x)\\
&[C\ e_1 \ldots e_n](\rho) = [e_1](\rho) \sqcap \ldots \sqcap [e_n](\rho), \text{ falls } C\ n\text{–stelliger Konstruktor mit } n > 0\\
&[f\ e_1 \ldots e_n](\rho) = [f]([e_1](\rho), \ldots, [e_n](\rho))
\end{aligned}$$

Dies ist auch die Form, die in der Literatur zur Notwendigkeitsanalyse vorgeschlagen wird.

Beispiel: [add]

$$[\![\texttt{Null},\texttt{y}]\!](\bot,N) = \{[\texttt{y} \mapsto \bot]\},\ [\![\texttt{Succ x},\texttt{y}]\!](\bot,N) = \{[\texttt{x} \mapsto \bot, \texttt{y} \mapsto \bot], [\texttt{x} \mapsto \bot, \texttt{y} \mapsto N]\}$$

$$[\texttt{add}]^{(0)}(d_1,d_2) = \bot \quad \forall\, d_1,d_2 \in \mathit{two}$$

$$[\texttt{add}]^{(1)}(\bot,N) = [\mathit{alt}_1]^{(1)}(\bot,N) \sqcup [\mathit{alt}_2]^{(1)}(\bot,N)$$

$$[\mathit{alt}_1]^{(1)}(\bot,N) = \bigsqcup_{\substack{\rho\in[\![\texttt{Null},\texttt{y}]\!](\bot,\texttt{N})\\ \rho \text{ maximal}}} [\texttt{y}](\rho) = \bot$$

$$\begin{aligned}[\mathit{alt}_1]^{(1)}(\bot,N) &= \bigsqcup_{\substack{\rho\in[\![\texttt{Succ x},\texttt{y}]\!](\bot,\texttt{N})\\ \rho \text{ maximal}}} [\texttt{Succ(add x y)}]^{(1)}(\rho)\\ &= [\texttt{Succ(add x y)}]^{(1)}(\underbrace{[\texttt{x} \mapsto \bot, \texttt{y} \mapsto N]}_{=\rho})\end{aligned}$$

$$[\texttt{Succ (add x y)}]^{(1)}(\rho) = [\texttt{add x y}]^{(1)}(\rho)$$

$$[\texttt{add}]^{(0)}([\texttt{x}]^{(1)}(\rho),[\texttt{y}]^{(1)}(\rho)) = [\texttt{add}]^{(0)}(\bot,N) = \bot$$

$$\Longrightarrow\ [\texttt{add}]^{(1)}(\bot,N) = \bot \sqcup \bot = \bot \qquad \text{usw.}$$

(2) *Beseitige den Operator* $\bigsqcup_{\rho\in[\![\underline{p}]\!](\underline{d})\ldots}$ *!*

Idee: Wir erfinden eine neue Funktion $[\![\underline{p}]\!] : \mathit{two}^n \longrightarrow (\mathit{Vars} \to \mathit{two})$ definiert durch

$$[\![\underline{p}]\!](\underline{d}) = \bigsqcup [\![\underline{p}]\!](\underline{d}).$$

D. h.: $([\![\underline{p}]\!](\underline{d}))(x) = N \iff \exists \rho \in [\![\underline{p}]\!](\underline{d}) : \rho(x) = N$
$\iff (x \in \mathit{Vars}(p_i) \Rightarrow d_i = N \vee \sharp\mathit{Vars}(p_i) > 1)$

Damit erhalten wir ein neues Gleichungssystem für $[\mathit{alt}]$:

$$[\mathit{alt}] = [e] \circ [\![\underline{p}]\!] \text{ d. h. } [\mathit{alt}](\underline{d}) = [e]([\![\underline{p}]\!](\underline{d}))$$

(3) *Wähle statt der Tabelle eine neue Datenstruktur !*

Repräsentiere die Funktionen nicht mehr als Wertetabellen, sondern mittels einer geeigneten Datenstruktur. Führe die Iterationen direkt auf dieser Datenstruktur aus !
Wegen $P(\mathit{two}) = \{\bot{\downarrow} \sqsubset N{\downarrow}\} \cong \{\bot \sqsubset N\} \cong \{0 \sqsubset 1\} = \mathit{two}'$ müssen wir **monotone** BOOLE'sche Funktionen repräsentieren.
Wir ordnen n–stelligen Funktionen f Werte in $[\mathit{two}'^n \to \mathit{two}']_m$ zu.
Mögliche Datenstrukturen:

- Disjunktive Normalformen:
  Für monotone Funktionen existieren eindeutige minimale DNF.
  Beispielsweise: $f(x_1,x_2,x_3) = x_1x_2 \vee x_2x_3 \vee x_1x_2x_3 = x_1x_2 \vee x_2x_3$
- OBDDs (**o**rdered **b**inary **d**ecision **d**iagrams)

Wir betrachten hier nur disjunktive Normalformen !

Wir benötigen jetzt eine Darstellung des Gleichungssystems, die direkt angibt, wie Funktionen aus anderen Funktionen konstruiert werden:

- $[f] = [\mathit{alt}_1] \sqcup \cdots \sqcup [\mathit{alt}_k] = [\mathit{alt}_1] \vee \cdots \vee [\mathit{alt}_k]$

- $[alt] = [e] \circ [\![\underline{p}]\!]$, wobei $[\![p_1, \ldots, p_n]\!] : \{0,1\}^n \longrightarrow (\mathit{Vars} \to \{0,1\})$ definiert ist durch

$$([\![p_1, \ldots, p_n]\!](x_1, \ldots, x_n))(y) := \begin{cases} \bigwedge\{x_i \mid \mathit{Vars}(p_i) = \emptyset\} \wedge x_j, & \text{falls } \{y\} = \mathit{Vars}(p_j) \\ \bigwedge\{x_i \mid \mathit{Vars}(p_i) = \emptyset\}^1, & \text{sonst} \end{cases}$$

- $[C] = \mathit{Const}_1 = 1$, falls C 0–stelliger Konstruktor
- $[x] = \pi_x = x$
- $[C\ e_1 \ldots e_n] = [e_1] \sqcap \cdots \sqcap [e_n] = [e_1] \wedge \cdots \wedge [e_n]$, falls C n–stelliger Konstruktor $(n > 0)$
- $[f\ e_1 \ldots e_n] = [f] \circ ([e_1] \times \cdots \times [e_n]) \circ \Delta_n = [f][[e_i]/x_i]$ (Substitution der Variablen)

Wir betrachten hier monotone Abbildungen aus $[\{0,1\}^n \to \{0,1\}]_m$ als mögliche Werte für $[f]$, bzw. monotone Abbildungen aus $[(\mathit{Vars} \to \{0,1\}) \to \{0,1\}]_m$ als Werte für $[e]$, repräsentiert als DNF.

Auf der Datenstruktur benötigten wir Operationen $\sqcap, \sqcup, \circ$ und Konstanten $1, \times, [\![\underline{p}]\!], \Delta_n$

Beispiel: $alt_1 \equiv$`a [] y = y` $\quad alt_2 \equiv$ `a (x:xs) y = x:(a xs y)`

$[\![\texttt{[]}, \texttt{y}]\!](x_1, x_2) = [\texttt{y} \mapsto x_1 x_2] =: \rho_1$
$[\![\texttt{x} : \texttt{xs}, \texttt{y}]\!](x_1, x_2) = [\texttt{x} \mapsto 1, \texttt{xs} \mapsto 1, \texttt{y} \mapsto x_2] =: \rho_2$

Gleichungssystem:

$$\begin{aligned}
&[\texttt{a}] = [alt_1] \vee [alt_2] \\
&[alt_1](x_1, x_2) = ([\texttt{y}] \circ [\![\texttt{[]}, \texttt{y}]\!])(x_1, x_2) = \texttt{y}(\rho_1) = x_1 x_2 \\
&[alt_2](x_1, x_2) \;=\; ([\texttt{x} : (\texttt{a xs y})] \circ [\![\texttt{x} : \texttt{xs}, \texttt{y}]\!])(x_1, x_2) = (\texttt{x} \wedge [\texttt{a xs y}])(\rho_2) \\
&\qquad\qquad\quad\;\; = \; 1 \wedge [\texttt{a xs y}](\rho_2) = [\texttt{a xs y}](\rho_2) \\
&\qquad\qquad\quad\;\; = \; ([\texttt{a}] \circ ([\texttt{xs}] \times [\texttt{y}]) \circ \Delta_2)(\rho_2) \\
&\qquad\qquad\quad\;\; = \; [\texttt{a}]([\texttt{xs}](\rho_2), [\texttt{y}](\rho_2)) = [\texttt{a}](1, x_2) \\
&\Longrightarrow [\texttt{a}](x_1, x_2) = x_1 x_2 \vee [\texttt{a}](1, x_2)
\end{aligned}$$

Fixpunktiteration:

$$\begin{aligned}
&[\texttt{a}]^{(0)}(x_1, x_2) = 0 \\
&[\texttt{a}]^{(1)}(x_1, x_2) = x_1 x_2 \vee [\texttt{a}]^{(0)}(1, x_2) = x_1 x_2 \vee 0 = x_1 x_2 \\
&[\texttt{a}]^{(2)}(x_1, x_2) = x_1 x_2 \vee [\texttt{a}]^{(1)}(1, x_2) = x_1 x_2 \;\vee\; 1 \wedge x_2 = x_1 x_2 \vee x_2 = x_2
\end{aligned}$$

Eine Lösung ist zwar ‘schnell’, d. h. einfach zu berechnen aber:
Achtung: Minimale disjunktive Normalformen n–stelliger Funktionen können exponentiell groß werden !

Um eine präzisere Lösung zu erhalten, müssen wir das Verfahren mit dem genaueren Patternmatching implementieren.

Satz:
Seien $\underline{p} = p_1 \ldots p_n$ und $\underline{d} = (d_1, \ldots, d_n) \in two^n$. Dann gilt:

1. $\rho \in [\![\underline{p}]\!](\underline{d}) \iff [\forall i : \bigwedge_{y \in \mathit{Vars}(p_i)} \rho(y) \to d_i]$

[1] wobei $\bigwedge \emptyset = 1$

2. Für monotones $\varphi : (\mathit{Vars} \to \mathit{two}) \to \mathit{two}$ ist $\varphi \odot [\![\underline{p}]\!] = \bigvee_{m \in \varphi} \bigwedge_{\mathit{Vars}(p_i) \subseteq m} x_i$
   ($m \in \varphi$ heißt, daß m ein Monom von φ ist)

Beweis:

zu 1.: $$\begin{aligned} \rho \in [\![\underline{p}]\!](\underline{d}) &\iff [d_i = 0 \to \exists y \in \mathit{Vars}(p_i) : \rho(y) = 0] \\ &\iff [\forall y \in \mathit{Vars}(p_i) : \rho(y) = 1 \to d_i = 1] \end{aligned}$$

zu 2.: $\varphi \odot [\![\underline{p}]\!](\underline{d}) = \bigsqcup_{\rho \in [\![\underline{p}]\!](\underline{d})} \varphi(\rho)$

d. h.

$$\begin{aligned} \varphi \odot [\![\underline{p}]\!](\underline{d}) = 1 \quad &\Longrightarrow \quad \exists \rho \in [\![\underline{p}]\!](\underline{d}) : \varphi(\rho) = 1 \\ & \qquad\ \ \exists \rho \in [\![\underline{p}]\!](\underline{d}) : \exists m \in \varphi : m(\rho) = 1 \\ &\Longrightarrow \quad \forall i \text{ mit } \mathit{Vars}(p_i) \subseteq m : d_i = 1 \\ &\Longrightarrow \quad \left(\bigvee_{m \in \varphi} \bigwedge_{\mathit{Vars}(p_i) \subseteq m} x_i \right) (\underline{d}) = 1 \end{aligned}$$

$$\left(\bigvee_{m \in \varphi} \bigwedge_{\mathit{Vars}(p_i) \subseteq m} x_i \right) (\underline{d}) = 1 \Longrightarrow \exists m \in \varphi : \forall i \text{ mit } \mathit{Vars}(p_i) \subseteq m : d_i = 1$$

Konstruiere $\rho : \mathit{Vars} \to \mathit{two}$ durch: $\rho(y) = \begin{cases} 1 & \text{falls } y \in m \\ 0 & \text{sonst} \end{cases}$.

Dann ist $\rho \in [\![\underline{p}]\!](\underline{d})$ und $m(\rho) = 1$. Folglich ist $\varphi \odot [\![\underline{p}]\!](\underline{d}) = 1$.

Damit erhalten wir als Gleichungssystem:

- $[f] = [\mathit{alt}_1] \vee \cdots \vee [\mathit{alt}_k]$
- $[\mathit{alt}] = \bigvee_{m \in [e]} \bigwedge_{\mathit{Vars}(p_i) \subseteq m} x_i$
- $[C] = 1$, falls C 0–stelliger Konstruktor
- $[x] = \pi_x = x$
- $[C\ e_1 \dots e_n] = [e_1] \wedge \cdots \wedge [e_n]$, falls C n–stelliger Konstruktor ($n > 0$)
- $[f\ e_1 \dots e_n] = [f]([e_1], \dots, [e_n])$

Beispiel: $\mathit{alt}_1 \equiv$`a [] y = y` $\qquad \mathit{alt}_2 \equiv$ `a (x:xs) y = x:(a xs y)`

$[\mathtt{a}] = [\mathit{alt}_1] \vee [\mathit{alt}_2]$

$[\mathit{alt}_1] = x_1 x_2$, da $[\mathtt{y}] = \mathtt{y}$ und $\mathit{Vars}(\mathtt{[]}) \subseteq \{\mathtt{y}\}$, $\mathit{Vars}(\mathtt{y}) \subseteq \{\mathtt{y}\}$

$[\mathit{alt}_2] = \bigvee_{m \in [\mathtt{x:(a\ xs\ y)}]} \bigwedge_{\mathit{Vars}(p_i) \subseteq m} x_i$

$[\mathtt{x} : (\mathtt{a\ xs\ y})] = x \wedge [\mathtt{a}](\mathtt{xs}, \mathtt{y})$

$[\mathit{alt}_2]^{(1)} = \bigvee_{m \in \mathtt{x} \wedge [\mathtt{a}]^{(0)}(\mathtt{xs},\mathtt{y})} \bigwedge_{\mathit{Vars}(p_i) \subseteq m} x_i = 0$

$[\mathtt{a}]^{(1)} = [\mathit{alt}_1] \vee [\mathit{alt}_2]^{(1)} = x_1 x_2$

$[\mathit{alt}_2]^{(2)} = \bigvee_{m \in x \wedge \mathtt{xs} \wedge \mathtt{y}} \bigwedge_{\mathit{Vars}(p_i) \subseteq m} x_i = x_1 x_2$ $\qquad$ d. h.

$[\mathtt{a}]^{(2)} = x_1 x_2 \vee x_1 x_2 = x_1 x_2$

3.5.4 Zusammenhang zwischen der Semantik über *two* und der Semantik über $\mathcal{T}$

Vorgehensweise:

- Wir führen eine Abstraktionsrelation $\triangle \subseteq \mathcal{T} \times two$ ein mit

$$t\triangle\bot :\iff t \text{ enthält } \bot$$
$$t\triangle N := True$$

- Fortsetzung auf der Relation $\triangle_P \subseteq P(\mathcal{T}) \times P(two)$ durch

$$A\triangle_P A^{\#} :\iff \forall\, a \in A\ \exists a^{\#} \in A^{\#} : a\triangle a^{\#}$$
d. h.$A\triangle_P \{\bot\} :\iff \forall\, a \in A$ gilt: a enthält $\bot$
$$A\triangle_P \{N,\bot\} = True$$

- Wir zeigen die Verträglichkeit mit allen Konstanten und Operationen unseres Gleichungssystems.

<u>Satz:</u>

1. Die Funktionen s, π_j und die Diagonale Δ_n sind verträglich mit $\triangle$ bzw. $\triangle_P$ (d. h. beispielsweise $d\triangle d^{\#} \ \Rightarrow\ s(d)\triangle_P\, s(d^{\#})$).
2. $[C]_{\mathcal{T}}\triangle[C]_{two}$ gilt für jeden Konstruktor C (der Index dient lediglich der Unterscheidung von Funktionen auf $\mathcal{T}$ bzw. *two*).
3. $[\![\underline{p}]\!]_{\mathcal{T}}\triangle[\![\underline{p}]\!]_{two}$ gilt für jede Patternliste $\underline{p} = p_1, \ldots, p_n$ (d. h. $\underline{d}\triangle\underline{d}^{\#} \ \Rightarrow\ [\![\underline{p}]\!]_{\mathcal{T}}(\underline{d})\triangle_P\, [\![\underline{p}]\!]_{two}(\underline{d}^{\#})$).
4. $\circ, \diamond, \odot$ und $\times$ sind ebenfalls verträglich mit $\triangle$ bzw. $\triangle_P$.

<u>Beweis:</u> Wir zeigen nur 3.
Seien $\underline{d} = (d_1, \ldots, d_n) \in \mathcal{T}^n$ und $\underline{d}^{\#} = d_1^{\#}, \ldots, d_n^{\#} \in two^n$.
Falls das Patternmatching fehlschlägt, ist $[\![\underline{p}]\!]_{\mathcal{T}}(\underline{d}) = \{\bot\} = \{\rho : x \mapsto \bot\ \forall x \in Vars\}$.
D. h. es gilt: $[\![\underline{p}]\!]_{\mathcal{T}}(\underline{d})\triangle_P A^{\#}\ \forall A^{\#} \in P(two)$
Bei erfolgreichem Patternmatching existiert ein $\rho : Vars \to \mathcal{T}$ mit $p_i\,\rho = d_i\ \forall 1 \le i \le n$
Sei nun $d_i\triangle d_i^{\#}\ \forall 1 \le i \le n$.
Zu zeigen: $[\![\underline{p}]\!]_{two}(\underline{d}^{\#})$ enthält ρ' mit $\rho\triangle\rho'$, d. h. es gilt $\rho(x)\triangle\rho'(x)\ \forall x \in Vars$.
Konstruiere $\rho^{\#}$ durch $\rho^{\#}(x) = \bot :\iff \rho(x)$ enthält $\bot$.
<u>Beh.:</u> $\rho^{\#} \in [\![\underline{p}]\!]_{two}(\underline{d}^{\#})$
Denn: Sei $d_i^{\#} = \bot$, dann enthält d_i $\bot$ (wegen $d_i\triangle d_i^{\#}$)
Sei $\underline{p} = p_1, \ldots, p_n$, dann ist $p_i\,\rho = d_i$.
$\implies \exists y \in Vars(p_i) : \rho(y)$ enthält $\bot \implies \rho^{\#}(y) = \bot$
Wir folgern: $\rho^{\#}$ erfüllt die Definition der Variablenbelegungen in $[\![\underline{p}]\!]_{two}(\underline{d}^{\#}) \implies$ Beh. □

Wir folgern weiterhin:

<u>Satz:</u> (Korrektheit der Abstraktion mittels *two*)
Seien $\eta_{\mathcal{T}}$ und η_{two} die kleinsten Lösungen von $[f] = F_f,\ f \in \mathcal{F}$ über $\mathcal{T}$ bzw. *two*. Dann gilt:
$$\eta_{\mathcal{T}}([f])\triangle\eta_{two}([f]) \text{ für alle } f \in \mathcal{F}.$$
(D. h.: $\underline{d}\triangle\underline{d}^{\#} \ \Rightarrow\ (\eta_{\mathcal{T}}[f])(\underline{d})\triangle_P(\eta_{two}[f])(\underline{d}^{\#})$)

<u>Beweis:</u> Fixpunktiteration

3.5.5 Zusammenhang zwischen der Semantik über *two* und der Notwendigkeitsanalyse

'Notwendigkeit' wird nun formalisiert als eine Eigenschaft.
Wir wollen eine 'Eigenschaft' möglicher Ergebnisse (nämlich kein $\perp$ zu enthalten) in eine Eigenschaft des Arguments transformieren.
Wir benötigen also eine Transformation die – beim Funktionsergebnis gestartet – Argumente liefert, die die gleichen Eigenschaften haben.

Definition:
Eine *Eigenschaft* in einer (geordneten) Menge D ist eine Teilmenge $O \subseteq D$, für die gilt

$$x \in O \text{ und } x \sqsubseteq d \implies d \in O$$

Beispiele:

- '*enthält nicht* $\perp$' ist eine Eigenschaft in `nat` (oder allgemein über $\mathcal{T}$).
- '≥ 5' ist eine Eigenschaft in $\{0 \sqsubset 1 \sqsubset 2 \sqsubset 3 \ldots\}$.
- '$\emptyset$' ist stets eine Eigenschaft.
- Sind O_i, $i \in I$ Eigenschaften in D, dann ist auch $\bigcup\limits_{i \in I} O_i$ eine Eigenschaft in D.

Definition:
$\Omega(D) := \{O \subseteq D \mid x \in O \implies x\uparrow \subseteq O\}$, bezeichnet die Menge der Eigenschaften in D, wobei $x\uparrow := \{y \in D \mid x \sqsubseteq y\}$.

$\Omega(D)$ ist ein vollständiger Verband mit der Mengeninklusion als Ordnung.
Im folgenden sei $s' : D \longrightarrow \Omega(D)$ die Einbettung $s'(d) = d\uparrow$.

Die Unterschiede zwischen Ω und $P(D)$ sind:

(a) Statt *nach unten abgeschlossenen* Mengen, betrachtet man *nach oben abgeschlossene* Mengen.

(b) $P(D)$ enthält im Gegensatz zu $\Omega(D)$ nicht die leere Menge.

Tatsächlich gilt für $\Omega(D)$ ein analoger Charakterisierungssatz wie für $P(D)$.

Sei $f : D_1 \longrightarrow P(D_2)$ monoton, wobei D_1 und D_2 ein kleinstes Element $\perp$ enthalten. Die Transformation von Eigenschaften von D_2 in Eigenschaften von D_1, die f bewirkt, ist gegeben durch:

$$f^- : \Omega(D_2) \longrightarrow \Omega(D_1) \text{ mit } f^-(O) = \{d_1 \in D_1 \mid f(d_1) \cap O \neq \emptyset\}$$

Satz:

1. f^- ist wohldefiniert und additiv mit $f^-(\emptyset) = \emptyset$ und $f^-(D_2) = D_1$.
2. $f_1 \sqsubseteq f_2 \implies f_1^- \sqsubseteq f_2^-$

Beweis:
zu 1.: Seien $d' \in f^-(O)$ und $d' \sqsubseteq d$.
f monoton $\Rightarrow f(d') \subseteq f(d) \Rightarrow f(d) \cap O \neq \emptyset \Rightarrow d \in f^-(O) \Rightarrow f^-$ ist wohldefiniert
$f^-(D_2) = \{d \in D_1 \mid f(d) \cap D_2 \neq \emptyset\} = D_1$

$f^-(\emptyset) = \{d \in D_1 \mid f(d) \cap \emptyset \neq \emptyset\} = \emptyset$

zu 2.: Sei $d \in f_1^-(O)$
Zu zeigen: $d \in f_2^-(O)$
Denn: Es gilt $f_1(d) \cap O \neq \emptyset$ und $f_1 \sqsubseteq f_2 \implies f_2(d) \cap O \neq \emptyset \implies d \in f_2^-(O)$ $\square$

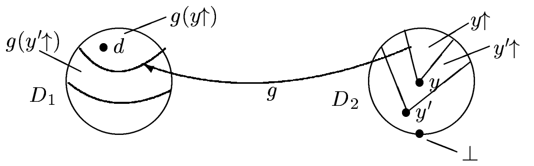
Bemerkung: Sei $[\Omega(D_2) \Rightarrow \Omega(D_1)]$ die Menge der additiven Funktionen $g : \Omega(D_2) \longrightarrow \Omega(D_1)$ mit $g(\emptyset) = \emptyset$ und $g(D_2) = D_1$ geordnet durch '$\sqsubseteq$'.
Die Transformation $(\cdot)^-$ vermittelt einen Ordnungsisomorphismus zwischen $[D_1 \to P(D_2)]_m$ und $[\Omega(D_2) \Rightarrow \Omega(D_1)]$.

Zum Beweis definieren wir eine inverse Transformation $(\cdot)^+$, die Funktionen aus $[\Omega(D_2) \Rightarrow \Omega(D_1)]$ in solche aus $[D_1 \to P(D_2)]_m$ zurückverwandelt.

Definition:
Sei $g \in [\Omega(D_2) \Rightarrow \Omega(D_1)]$. Dann ist $g^+ : D_1 \longrightarrow P(D_2)$ gegeben durch:

$$g^+(d) = \{y \in D_2 \mid d \in g(y\!\uparrow)\}, \text{ wobei } y\!\uparrow := \{y' \in D_2 \mid y \sqsubseteq y'\}.$$

$g^+(d)$ liefert tatsächlich eine Abbildung $D_1 \to P(D_2)$, die monoton ist.
Im Bild:



Satz:

1. Für monotones $f : D_1 \longrightarrow P(D_2)$ ist $(f^-)^+ = f$.
2. Für additives $g : \Omega(D_2) \longrightarrow \Omega(D_1)$ ist $(g^+)^- = g$.

Beweis:

$$\begin{array}{llcl}
\text{zu 1.:} & (f^-)^+(x) & = & \{y \in D_2 \mid x \in f^-(y\!\uparrow)\} \\
& & = & \{y \in D_2 \mid x \in \{d \in D_1 \mid f(d) \cap y\!\uparrow \neq \emptyset\}\} \\
& & = & \{y \in D_2 \mid f(x) \cap y\!\uparrow \neq \emptyset\} \\
& & = & \{y \in D_2 \mid y \in f(x)\} \\
& & = & f(x) \\
\text{zu 2.:} & (g^+)^-(O) & = & \{x \in D_1 \mid g^+(x) \cap O \neq \emptyset\} \\
& & = & \{x \in D_1 \mid \{y \in D_2 \mid x \in g(y\!\uparrow)\} \cap O \neq \emptyset\} \\
& & = & \{x \in D_1 \mid \{y \in O \mid x \in g(y\!\uparrow)\} \neq \emptyset\} \\
& & = & \{x \in D_1 \mid \exists y \in O : x \in g(y\!\uparrow) \\
& & = & \bigcup\limits_{y \in O} g(y\!\uparrow) \overset{\text{g add.}}{=} g(\bigcup\limits_{y \in O} y\!\uparrow) = g(O)
\end{array}$$
$\square$

Sei $\triangle \subseteq D \times D^\#$ eine Abstraktionsrelation.
Wir definieren $\triangle_\Omega \subseteq \Omega(D) \times \Omega(D^\#)$ durch

$$O \triangle_\Omega O^\# :\iff \forall\, o \in O,\, o^\# \in D^\# :\; o \triangle o^\# \implies o^\# \in O^\#$$

Beispiele für $\triangle_\Omega$:

- konkrete Eigenschaft $\mathcal{N} = \{t \in \mathcal{T} \mid t \text{ enthält nicht } \perp\}$
  Es gilt: $\mathcal{N} \triangle_\Omega \{N, \perp\}$, aber auch $\mathcal{N} \triangle_\Omega \{N\}$.

- abstrakter Bereich *three* $= \{\bot \sqsubset H \sqsubset N\}$ mit
  $t \triangle \bot :\iff t = \bot$
  $t \triangle H :\iff t$ enthält $\bot$
  $t \triangle N := \mathit{True}$

  Es gilt: $\mathcal{N} \triangle_\Omega \{N\}$
  Für $\mathcal{H} := \{t \mid t \neq \bot\}$ gilt: $\mathcal{H} \triangle_\Omega \{H, N\}$, aber nicht: $\mathcal{H} \triangle_\Omega \{N\}$

- Seien $I\!N = \{0 \sqsubset 1 \sqsubset 2 \sqsubset \cdots\}$, $\mathbf{3} = \{0 \sqsubset 1 \sqsubset 2 \sqsubset 3 \sqsubset \infty\}$ und $\triangle \subseteq I\!N \times \mathbf{3}$ die Relation mit $n \triangle i :\iff n \sqsubseteq i$. Sei nun *zwei* $\subseteq I\!N$ die Eigenschaft *zwei* $= \{2, 3, 4, \ldots\}$.
  Dann wird *zwei* abstrahiert von $\{2, 3, \infty\}$ (und allen größeren Teilmengen von $\Omega(\mathbf{3})$), aber nicht von $\{3, \infty\}$.

Bleibt noch zu folgern: $f \triangle f^\# \overset{?}{\Longrightarrow} f^- \triangle_\Omega f^{\#-}$
$\overset{?}{\Longrightarrow}$ gilt zumindest dann, wenn $\triangle$ eine 'beste Approximation' erlaubt !

Beste Approximationen:

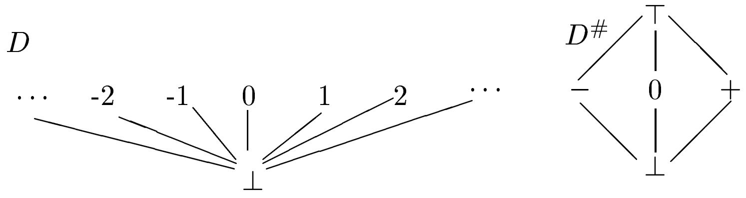
Definition:
Seien $\triangle \subseteq D \times D^\#$ eine Abstraktionsrelation, $D^\#$ ein vollständiger Verband und $\alpha : D \longrightarrow D^\#$ die Abbildung mit $\alpha(x) = \bigsqcap \{d \in D^\# \mid x \triangle d\}$.
Falls für alle $x \in D$ gilt: $x \triangle \alpha(x)$, so nennen wir α die *beste Approximation* (relativ zu $\triangle$) und sagen: $\triangle$ erlaubt eine beste Approximation.

Beachte: α ist monoton !
Beste Approximationen heißen auch *Abstraktionsfunktionen*.

Beispiel:

- Für $\alpha(t) = \begin{cases} \bot, & \text{falls } t \ \bot \text{ enthält} \\ N, & \text{sonst} \end{cases}$ erlaubt $\triangle \subseteq \mathcal{T} \times \mathit{two}$ eine beste Approximation.

- Für



ist α mit $\alpha(d) = \begin{cases} \bot, & \text{falls } d = \bot \\ -, & \text{falls } d < 0 \\ 0, & \text{falls } d = 0 \\ +, & \text{sonst} \end{cases}$ beste Approximation relativ zu $\triangle \subseteq D \times D^\#$

Satz: (Eigenschaften bester Approximationen)

1. Erlauben $\triangle_i \subseteq D_i \times D_i^\#$ beste Approximationen α_i, dann auch $\triangle \subseteq \triangle_1 \times \triangle_2$, wobei $(d_1, d_2) \triangle (d_1^\#, d_2^\#) \iff d_i \triangle_i d_i^\#$ für $i = 1, 2$. Dabei ist die beste Approximation α gegeben durch $\alpha = \alpha_1 \times \alpha_2$.

2. Sei α eine beste Approximation zu $\triangle \subseteq D \times D^\#$. Dann ist $P(s \circ \alpha) : P(D) \longrightarrow P(D^\#)$ eine beste Approximation für $\triangle_P \subseteq P(D) \times P(D^\#)$.

$$\begin{tikzcd}
D \arrow[r, "\alpha"] \arrow[d, "s"'] \arrow[dr, "s\circ\alpha"] & D^{\#} \arrow[d, "s"] \\
P(D) \arrow[r, dotted, "P(s\circ\alpha)"] & P(D^{\#})
\end{tikzcd}$$

Analog gibt es für $\triangle_\Omega$ eine beste Approximation definiert durch: $\Omega(s' \circ \alpha)$, wobei

$$(\Omega(s' \circ \alpha))(O) := \bigcup_{o \in O} s'(\alpha(o)) = \bigcup_{o \in O} (\alpha(o))\uparrow.$$

$$\begin{tikzcd}
D \arrow[r, "\alpha"] \arrow[d, "s'"'] \arrow[dr, "s'\circ\alpha"] & D^{\#} \arrow[d, "s'"] \\
\Omega(D) \arrow[r, dotted, "\Omega(s'\circ\alpha)"] & \Omega(D^{\#})
\end{tikzcd}$$

3. Besitzen $\triangle \subseteq D \times D^{\#}$ und $\triangle' \subseteq \overline{D} \times \overline{D}^{\#}$ beste Approximationen α und α', dann gilt für monotone Funktionen $f : D \longrightarrow \overline{D}$ bzw. $f^{\#} : D^{\#} \longrightarrow \overline{D}^{\#}$:

   $$f \triangle f^{\#} \iff \alpha' \circ f \sqsubseteq f^{\#} \circ \alpha$$

   (wobei $f \triangle f^{\#} \iff \forall\, d \triangle d^{\#} : f(d) \triangle' f^{\#}(d^{\#})$)

   Im Bild:

   $$\begin{tikzcd}
   D \arrow[r, "f"] \arrow[d, "\alpha"'] & \overline{D} \arrow[d, "\alpha'", "\sqsubseteq"'] \\
   D^{\#} \arrow[r, "f^{\#}"] & \overline{D}^{\#}
   \end{tikzcd}$$

<u>Beweis:</u>

zu 2. für $\triangle_P$:

$(P(s \circ \alpha))(A) = \bigcup_{x \in A} (\alpha(x))\downarrow = \bigcup_{x \in A} \{y \in D^{\#} \mid y \sqsubseteq \alpha(x)\}$

Wir zeigen: (a) $A \triangle_P A^{\#} \implies (P(s \circ \alpha))(A) \subseteq A^{\#}$ für alle $A \in P(D), A^{\#} \in P(D^{\#})$

(b) $A \triangle_P (P(s \circ \alpha))(A)$ für alle $A \in P(D)$

Zu (a): $A \triangle_P A^{\#} \implies \forall\, x \in A\ \exists x^{\#} \in A^{\#} : x \triangle x^{\#}$
$\implies \forall\, x \in A : \alpha(x) \in A^{\#}$
$\implies \forall\, x \in A : \{y \in D^{\#} \mid y \sqsubseteq \alpha(x)\} \subseteq A^{\#}$
$\implies (P(s \circ \alpha))(A) \subseteq A^{\#}$

Zu (b): Zu jedem $x \in A$ finden wir ein $x^{\#} \in (P(s \circ \alpha))(A) \subseteq A^{\#}$ mit $x \triangle x^{\#}$, wie etwa $x^{\#} := \alpha(x)$.

Für $\triangle_\Omega$:

Wir zeigen: (a) $O \triangle_\Omega O^{\#} \implies (\Omega(s' \circ \alpha))(O) \subseteq O^{\#}$ für alle $O \in \Omega(D), O^{\#} \in \Omega(D^{\#})$

(b) $O \triangle_\Omega (\Omega(s' \circ \alpha))(O)$ für alle $O \in \Omega(D)$

Zu (a): $O \triangle_\Omega O^{\#} \implies \forall\, x \in O\ \forall\, x^{\#} \in D^{\#} : x \triangle x^{\#} \Rightarrow x^{\#} \in O^{\#}$
$\implies \forall\, x \in O : \alpha(x) \in O^{\#}$
$\implies (\Omega(s' \circ \alpha))(O) \subseteq O^{\#}$

Zu (b): Sei $x \in O$ und $x \triangle x^{\#}$

Dann gilt: $\alpha(x) \sqsubseteq x^{\#} \implies x^{\#} \in (\Omega(s' \circ \alpha))(O)$, da $(\alpha(x))\uparrow \subseteq (\Omega(s' \circ \alpha))(O)$.

zu 3.:

$$
\begin{array}{lll}
\text{“}\Rightarrow\text{”}: x\triangle\alpha(x) & \Longrightarrow & f(x)\triangle f^{\#}(\alpha(x)) \\
 & \Longrightarrow & \alpha'(f(x)) \sqsubseteq f^{\#}(\alpha(x)) \\
\text{“}\Leftarrow\text{”}: x\triangle x^{\#} & \Longrightarrow & \alpha(x) \sqsubseteq x^{\#} \\
 & \Longrightarrow & \alpha'(f(x)) \overset{\text{nach Vor.}}{\sqsubseteq} f^{\#}(\alpha(x)) \overset{\text{Mon.}}{\sqsubseteq} f^{\#}(x^{\#}) \\
 & \Longrightarrow & f(x)\triangle f^{\#}(x^{\#}) \qquad \square
\end{array}
$$

Theorem:
Seien $f : D_1 \longrightarrow P(D_2)$, $f^{\#} : D_1^{\#} \longrightarrow P(D_2^{\#})$ monotone Abbildungen und $\alpha_i : D_i \longrightarrow D_i^{\#}$, $i = 1, 2$ beste Approximationen (für $\triangle_i \subseteq D_i \times D_i^{\#}$). Dann gilt:

$f\triangle f^{\#} \Longrightarrow f^{-}\triangle_\Omega f^{\#-}$, d. h. $O\triangle_\Omega O^{\#} \Longrightarrow f^{-}(O)\triangle_\Omega f^{\#-}(O^{\#})$

Beweis:
Sei $\alpha_i' = \Omega(s' \circ \alpha_i)$, $i = 1, 2$.
Zu zeigen: $\alpha_1'(f^{-}(O)) \subseteq f^{\#-}(\alpha_2'(O))$ für alle $O \in \Omega(D_2)$

$$
\begin{array}{lll}
\alpha_1'(f^{-}(O)) & = & \alpha_1'(\{x \in D_1 \mid f(x) \cap O \neq \emptyset\}) \\
 & \subseteq & \alpha_1'(\{x \in D_1 \mid P(s \circ \alpha_2)(f(x)) \cap \alpha_2'(O) \neq \emptyset\}) \quad (*) \\
 & \subseteq & \alpha_1'(\{x \in D_1 \mid f^{\#}(\alpha_1(x)) \cap \alpha_2'(O) \neq \emptyset\}) \\
 & = & \{x \in D_1^{\#} \mid \alpha_1(x) \sqsubseteq x^{\#} \text{ und } f^{\#}(\alpha_1(x)) \cap \alpha_2'(O) \neq \emptyset\} \\
 & \subseteq & \{d^{\#} \in D_1^{\#} \mid f^{\#}(d^{\#}) \cap \alpha_2'(O) \neq \emptyset\} \\
 & = & f^{\#-}(\alpha_2'(O))
\end{array}
$$

Dabei gilt $(*)$ wegen: $y \in f(x) \cap O \Longrightarrow \alpha_2(y) \in P(s \circ \alpha_2)(f(x))$ und außerdem $\alpha_2(y) \in \alpha_2'(O) \Longrightarrow \alpha_2(y) \in P(s \circ \alpha_2)(f(x)) \cap \alpha_2'(O)$ $\square$.

Fazit: Wir können wie folgt vorgehen:

(1) Definiere zu $\mathcal{T}$ eine beste Approximation $\alpha : \mathcal{T} \longrightarrow D$ für einen geeigneten abstrakten Bereich D.

(2) Berechne die abstrakte denotationelle Semantik über D (möglich,sofern $\sharp D < \infty$). Dann gilt: $[f]_{\mathcal{T}}\triangle[f]_D$ für alle $f \in \mathcal{F}$.

(3) Berechne zu einer Eigenschaft $P \in \Omega(\mathcal{T})$ die abstrakte Eigenschaft $P^{\#} = \Omega(s' \circ \alpha)(P) \in \Omega(D)$.

(4) Für jedes $f \in \mathcal{F}$ berechne $[f]_D^{-}(P^{\#})$. Dann gilt: $[f]_{\mathcal{T}}^{-}(P)\triangle_\Omega[f]_D^{-}(P^{\#})$ oder andersherum $\alpha(x) \notin [f]_D^{-}(P^{\#}) \Longrightarrow x \notin [f]_{\mathcal{T}}^{-}(P)$.

Beispiel:

Es sei $\mathcal{N} = \{t \in \mathcal{T} \mid t \text{ enthält nicht } \perp\}$ eine Eigenschaft und $D = \mathit{two}$ mit $\mathcal{N}\triangle\{N\}$.

$$
\begin{array}{lll}
[f]_{\mathit{two}}^{-}(\{N\}) & = & \{\underline{d} \in \mathit{two}^n \mid [f]_{\mathit{two}}(\underline{d}) \cap \{N\} \neq \emptyset\} \\
 & = & \{\underline{d} \in \mathit{two}^n \mid N \in [f]_{\mathit{two}}(\underline{d})
\end{array}
$$

Die Aussage 'j–tes Argument wird **ganz** benötigt' heißt:

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\forall\, (t_1, \ldots, t_n) \in \mathcal{T}^n \text{ gilt} : t_j \text{ enthält } \perp \Rightarrow (t_1, \ldots, t_n) \notin [f]_{\mathcal{T}}^{-}(\mathcal{N})]} \\
\Longleftarrow & [\forall\, (t_1, \ldots, t_n) \in \mathcal{T}^n \text{ gilt} : \alpha(t_j) = \perp \Rightarrow (\alpha(t_1), \ldots, \alpha(t_n)) \notin [f]_{\mathit{two}}^{-}(\{N\})] \\
\Longleftarrow & [\forall\, (d_1, \ldots, d_n) \in \mathcal{T}^n \text{ gilt} : d_j = \perp \Rightarrow (d_1, \ldots, d_n) \notin [f]_{\mathit{two}}^{-}(\{N\})] \\
\Longleftrightarrow & (N, \ldots, N, \underbrace{\perp}_{\text{Pos. } j}, N, \ldots, N) \notin [f]_{\mathit{two}}^{-}(\{N\}) \\
\Longleftrightarrow & [f]_{\mathit{two}}(N, \ldots, N, \underbrace{\perp}_{\text{Pos. } j}, N, \ldots, N) = \perp\downarrow
\end{array}
$$

Beispielsweise erhält man für die Append–Funktion a durch ungenauere Analyse $[\mathsf{a}]_{two} = x_2$ (mit Monomen notiert), wobei $P(two) = \{0 \sqsubset 1\}$
$\Longrightarrow$ a benötigt das zweite Argument ganz.
Eine genauere Analyse liefert: $[\mathsf{a}]_{two} = x_1 \wedge x_2$
$\Longrightarrow$ a benötigt sowohl das erste, als auch das zweite Argument ganz, sofern das Ergebnis ganz benötigt wird.

<u>Vergleich der komplizierten Analyse zur einfachen Analyse am Anfang:</u>

Definiere eine Abstraktionsfunktion $\alpha : [\{0,1\}^n \to \{0,1\}]_m \longrightarrow \mathcal{P}(\{1,\ldots,n\})$ durch:

$$\alpha(\varphi) := \{j \mid \varphi(1,\ldots,1,\underbrace{0}_{\text{Pos. } j},1,\ldots,1) = 0\}$$

Insbesondere ist

$$\begin{aligned} &\alpha(x_1 \wedge \cdots \wedge x_n) = \{1,\ldots,n\} \\ &\alpha\left(\bigwedge_{j\in A} x_j\right) = A, \quad A \subseteq \{1,\ldots,n\} \\ \text{und} \quad &\alpha(0) = \{1,\ldots,n\} \text{ und } \alpha(1) = \emptyset \end{aligned}$$

Kapitel 4

Analyse logischer Programme

4.1 Einführung

1.) Datenobjekte:
Hierbei handelt es sich um Konstruktorterme, die evtl. freie Variablen enthalten, z. B.:
`a(X,Y)`, `1:[]` (Listen–Konstruktor) oder `1:(2:X)`
Achtung: Variablen werden **groß** geschrieben. Konstruktoren werden **klein** geschrieben und müssen nicht deklariert werden (für `1:[]` schreibt man auch `[1]`).

2.) Prädikate:
Prädikate auf Tupeln von Termen sind durch Regeln definiert:
Beispiel:

`app([],Y,Y)`$\leftarrow$ (eine solche Regel ohne Voraussetzung heißt Fakt)
`app(X:XS,Y,X:Z)`$\leftarrow$`app(XS,Y,Z)`

oder

$$\left.\begin{array}{l}\texttt{father(fritz,leo)} \leftarrow \\ \texttt{mother(anca,leo)} \leftarrow \\ \texttt{father(ecki,anca)} \leftarrow\end{array}\right\}\text{ Fakten}$$

`ancestor(X,Y)`$\leftarrow$`father(X,Y)`
`ancestor(X,Y)`$\leftarrow$`mother(X,Y)`

$$\underbrace{\texttt{ancestor(X,Y)}}_{\textit{Kopf}\text{ der Regel}} \leftarrow \underbrace{\texttt{ancestor(X,Z)}, \underbrace{\texttt{ancestor(Z,Y)}}_{\textit{Ziel}}}_{\textit{Rumpf}\text{ der Regel}}$$

3.) Anfragen:
Die Programmausführung startet mit einer Anfrage. Z.B.:
? `app([1],[2,3],[1,2,3])`
? `app([1],[2,3],Z)`
? `app(X,[2,3],[1,2,3])`
? `ancestor(ecki,Y)`

Die Grammatik für logische Programme:

$$\begin{array}{llll} pr & ::= & rs\ ?\ g & \text{(Programm)} \\ rs & ::= & \varepsilon \mid r; rs & \text{(Regelmenge)} \\ r & ::= & g_0 \leftarrow g_1, \ldots, g_m & \text{(Regel)} \\ g & ::= & p(t_1, \ldots, t_n) & \text{(Goal, Ziel, Anfrage)} \\ t & ::= & a \mid X \mid a(t_1, \ldots, t_n) & \text{(Term)} \end{array}$$

Ziel einer Berechnung ist, alle Substitutionen zu finden, die die Anfrage 'wahr' machen, d. h. mit Hilfe der Regeln aus den Fakten deduzierbar sind.

Beispiel:

```
? app([1],[2,3],[1,2,3]) – yes
? app([1],[2,3],Z) – Z=[1,2,3]
? app(X,[2,3],[1,2,3]) – X=[1]
? app(X,Y,[1,2,3]) – X=[],Y=[1,2,3]
                   – X=[1],Y=[2,3]
                   – X=[1,2],Y=[3]
                   – X=[1,2,3],Y=[]
```

Es kann also durchaus mehrere Ergebnis–Substitutionen geben.

Achtung:

1. I. a. kann die Ergebnissubstitution auch Variablenvorkommen enthalten.
   Z. B.: ? `app([1],Y,[1,T,3])` – `Y=[T,3]`
   Dies ist offenbar eine sinnvolle Abkürzung für die Menge $\{\mathtt{y} \mapsto \mathtt{[t,3]}, \mathtt{T} \mapsto \mathtt{t} \mid \mathtt{t} \text{ ist Term}\}$, die alle möglichen Ergebnissubstitutionen darstellt.
   $\sigma_0 = [\mathtt{Y} \mapsto \mathtt{[T,3]}]$ ist unter diesen eine *allgemeinste* Substitution, d. h. alle anderen Substitutionen aus dieser Menge erhält man, indem man eine Komposition $\sigma_1 \circ \sigma_0$ konstruiert ($\sigma_1 = [\mathtt{T} \mapsto \mathtt{t}]$).
   Wir schreiben für $\sigma_1 \circ \sigma_0$ auch abkürzend $\sigma_1\sigma_0$ und bezeichnen die leere Substitution mit $\emptyset$.

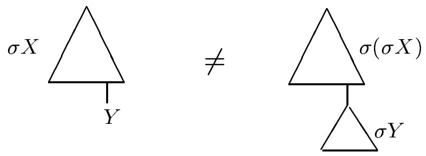
2. Mögliche Ergebnissubstitutionen σ bilden alle Variablen X, für die nichts angegeben ist, auf X selbst ab. Außerdem betrachten wir nur *idempotente* Substitutionen, d. h. es gilt stets: $\sigma \circ \sigma = \sigma$.
   Im Beispiel:
   Für $\sigma_1 = [\mathtt{X} \mapsto \mathtt{c}(\mathtt{Y}, \mathtt{a}), \mathtt{Y} \mapsto \mathtt{b}(\mathtt{Z})]$ ist $\sigma_2 = \sigma_1 \circ \sigma_1 = [\mathtt{X} \mapsto \mathtt{c}(\mathtt{b}(\mathtt{Z}), \mathtt{a}), \mathtt{Y} \mapsto \mathtt{b}(\mathtt{Z})]$.
   Wegen $\sigma_2 \neq \sigma_1$ ist σ_1 nicht imdempotent, aber σ_2 ist idempotent.

Satz:
Sei $dom(\sigma) = \{X \mid \sigma X \neq X\}$. Dann gilt:

$$\sigma \circ \sigma = \sigma \iff \forall X \in dom(\sigma) : Vars(\sigma X) \cap dom(\sigma) = \emptyset$$

Beweis:
"$\Leftarrow$": $X \notin dom(\sigma) \implies \sigma X = X \implies \sigma(\sigma X) = X = \sigma X$
$X \in dom(\sigma) \implies \sigma(\sigma X) = \sigma X$ da σX keine Variable aus $dom(\sigma)$ enthält

"⇒": Annahme: Es gebe ein $X \in dom(\sigma)$ mit $Vars(\sigma X) \cap dom(\sigma) \neq \emptyset$. Sei $Y \in Vars(\sigma X) \cap dom(\sigma)$.
Dann ist $\sigma Y \neq Y$. Y ist zwar in σX enthalten, aber nicht in $\sigma(\sigma X)$.



$\Longrightarrow$ Widerspruch zu $\sigma = \sigma \circ \sigma$ □

4.2 Unifikation

Definition:
Ein *allgemeinster Unifikator (most general unifier (mgu))* zweier Terme t_1, t_2 ist eine idempotente Substitution σ, für die gilt:

1. $\sigma t_1 = \sigma t_2$
2. Für jede andere idempotente Substitution σ' mit $\sigma' t_1 = \sigma' t_2$ ist $\sigma' = \sigma'' \circ \sigma$ für eine idempotente Substitution σ''.

Beispiele:

- $t_1 =$`a(X,b(a(c,c)))` und $t_2 =$`a(b(Z),Z)`
  $mgu(t_1, t_2) = [\mathtt{Z} \mapsto \mathtt{b(a(c,c))}, \mathtt{X} \mapsto \mathtt{b(b(a(c,c)))}]$
- $t_1 =$`a(c,b(Z))` und $t_2 =$`a(d,X)`
  Diese beiden Terme besitzen keinen *mgu*.
- $t_1 = \mathtt{X}$ und $t_1 = \mathtt{Y}$
  Mögliche *mgu*'s sind: $[\mathtt{X} \mapsto \mathtt{Y}]\,[\mathtt{Y} \mapsto \mathtt{X}]$ und $[\mathtt{X} \mapsto \mathtt{Z}, \mathtt{Y} \mapsto \mathtt{Z}]$ usw.
  Alle diese *mgu*'s sind aber 'äquivalent', d. h. sie gehen durch Variablenumbenennung in den rechten Seiten auseinander hervor.

Konvention: Wir betrachten *mgu*'s, die nur Variablen aus $Vars(t_1) \cup Vars(t_2)$ substituieren und haben im 'Bild' der Substitution ebenfalls nur solche Variablen.
Dabei definieren wir das *Bild* $im(\sigma)$ *einer Substitution* durch:

$$im(\sigma) := \bigcup \{\, Vars(\sigma X) \mid \sigma X \neq X \}$$

Formal: $im(mgu(t_1, t_2)) \subseteq Vars(t_1) \cup Vars(t_2)$

Algorithmus zur Bestimmung eines *mgu*:

```
subst Mgu(term t, term s) {
  switch (t, s) {
    case (X,X): return []
    case (X,s): if (X ∉ Vars(s))        (*)
                  return [X ↦ s];
                else
                  return FAIL;
    case (t,X): if (X ∉ Vars(t))        (*)
                  return [X ↦ t];
                else
                  return FAIL;
    case (a,a): return [];
    case (a(t1,...,tn),a(s1,...,sn)):
        return Mgulist((t1,s1),...,(tn,sn));
    default: return FAIL;
    }   /* switch */
}

subst Mgulist(list_of_termpairs l ) {
subst σ, σ';
  σ := [];
  repeat
    (t,s) :=head(l);
    σ' :=Mgu(σt,σs);
    if (σ' == FAIL) return FAIL;
    σ := σ' ∘ σ;
  until empty(l);
  return σ;
}
```

Dabei liefert **head** das erste Element der Liste und entfernt es, **empty** überprüft, ob die Liste leer ist.
Die mit $(*)$ markierten Zeilen sind der sogenannte *Vorkommenstest (occur check)* für Variablen. Dieser stellt fest, ob die Variable `X` ebenfalls im Term t (bzw. s) vorkommt (offenbar besitzen Terme wie etwa `X` und `a(X)` keinen **endlichen** allgemeinsten Unifikator). Der Vorkommenstest wird oft überhaupt nicht durchgeführt, weil er zu 'teuer' ist !

Berechnet **Mgu** eine Substitution σ, die verschieden von *FAIL* ist, so ist σ idempotent! (Beweis siehe Übung)

Wir sprechen deshalb im folgenden auch von **dem** *mgu* zweier Terme. Damit meinen wir den *mgu*, den unser Algorithmus berechnet.

Das Konzept der Unifikation kann auch auf Substitutionen fortgesetzt werden.

Definition:
Ein (bzw. der) allgemeinster Unifikator zweier idempotenter Substitutionen σ_1, σ_2 ist eine idempotente Substitution σ, für die gilt:

1. $\sigma \circ \sigma_1 = \sigma \circ \sigma_2$
2. Gilt $\sigma' \circ \sigma_1 = \sigma' \circ \sigma_2$, dann gibt es ein $\bar{\sigma}$ mit $\sigma' = \bar{\sigma} \circ \sigma$.

3. $im(\sigma) \subseteq im(\sigma_1) \cup im(\sigma_2)$

(Die Konvention in 3. wurde zusätzlich in die Definition eingebaut.)

Seien $\{X \mid \sigma_i X \neq X\} \subseteq \{X_1, \ldots, X_n\}$ für $i = 1, 2$.
Dann läßt sich der mgu von σ_1 und σ_2 berechnen durch:
$mgu(\sigma_1, \sigma_2) =$**Mgulist** $((\sigma_1 X_1, \sigma_2 X_1), \ldots (\sigma_1 X_n, \sigma_2 X_n))$

Achtung: Sind σ_1 und σ_2 idempotent, so muß $\sigma_2 \circ \sigma_1$ keineswegs idempotent sein.

Beispiel:

Seien $\sigma_1 = [\mathtt{X} \mapsto \mathtt{c(Y)}]$ und $\sigma_2 = [\mathtt{Z} \mapsto \mathtt{X}]$
Sowohl σ_1 als auch σ_2 sind zwar idempotent, aber $\sigma_2 \circ \sigma_1 = [\mathtt{X} \mapsto \mathtt{c(Y)}, \mathtt{Z} \mapsto \mathtt{X}]$ ist nicht idempotent!

Sei $dom(\sigma) := \{X \mid \sigma X \neq X\}$.

Satz:
Seien σ_1 und σ_2 idempotente Substitutionen, dann gilt:

1. $\sigma_2 \circ \sigma_1$ ist idempotent, wenn $im(\sigma_2) \cap dom(\sigma_1) = \emptyset$.
2. Sei $mgu(\sigma_1, \sigma_2)$ definiert (d. h. $\neq$*FAIL*).
   Dann ist $mgu(\sigma_1, \sigma_2) \circ \sigma_1$ idempotent, wenn $im(\sigma_2) \cap dom(\sigma_1) = \emptyset$.

Beweis:
zu 1.:
Wegen des vorherigen Satzes gilt: $\sigma_2 \circ \sigma_1$ idempotent $\iff im(\sigma_2 \circ \sigma_1) \cap dom(\sigma_2 \circ \sigma_1) = \emptyset$
Dabei ist $dom(\sigma_2 \circ \sigma_1) \subseteq dom(\sigma_1) \cup dom(\sigma_2)$
Weiterhin ist $im(\sigma_2 \circ \sigma_1) \subseteq im(\sigma_2) \cup (im(\sigma_1) \setminus dom(\sigma_2))$
Daraus folgt:
$im(\sigma_2 \circ \sigma_1) \cap dom(\sigma_2 \circ \sigma_1) \subseteq (im(\sigma_2) \cup (im(\sigma_1) \setminus dom(\sigma_2))) \cap (dom(\sigma_1) \cup dom(\sigma_2))$

$$\begin{array}{rll} = & (im(\sigma_2) \cap dom(\sigma_1)) & (1) \\ & \cup\ (im(\sigma_2) \cap dom(\sigma_2)) & (2) \\ & \cup\ ((im(\sigma_1) \setminus dom(\sigma_2)) \cap dom(\sigma_1)) & (3) \\ & \cup\ ((im(\sigma_1) \setminus dom(\sigma_2)) \cap dom(\sigma_2)) & (4) \end{array}$$

Die Durchschnitte (2) und (3) sind leer, da σ_1, σ_2 idempotent, Durchschnitt (4) ist leer per def. und (1) ist leer nach Voraussetzung. $\Longrightarrow$ Beh. $\square$

zu 2.:
Wegen 1. genügt es zu zeigen: $im(mgu(\sigma_1, \sigma_2)) \cap dom(\sigma_1) = \emptyset$
Per def. ist $im(mgu(\sigma_1, \sigma_2)) \subseteq im(\sigma_1) \cap im(\sigma_2) \quad \Longrightarrow$ Beh.

Vorgehensweise bei der Berechnung einer Ergebnissubstitution:

Idee: Sukzessives Reduzieren von Anfragen auf (hoffentlich) einfachere Anfragen unter Anwendung der Regeln. Eine Anwendung ist nur möglich, falls die Variablen bestimmten 'Restriktionen' unterliegen. Diese Restriktionen sammeln wir auf.
Wir gehen also wie folgt vor:

- Auswahl einer Regel r_k
- Umbenennung der Variablen in r_k

- Berechnung des allgemeinsten Unifikators von Anfrage und Kopf der Regel r_k, d. h. $\sigma_i = mgu$(Anfrage,Kopf)
- Restriktion von σ_i auf die Variablen der Regel r_k
- Abarbeiten der Ziele im Rumpf der Regel

Beispiel:

$r_1 \equiv$ `app([],YS,YS)`$\leftarrow$
$r_2 \equiv$ `app(X:XS,YS,X:ZS)`$\leftarrow$`app(XS,YS,ZS)`

σ $\boxed{\texttt{app(XS}_0\texttt{,[2,3],[1,2,3])}}$ σ'

r_2 $\equiv \texttt{app(X}_1\texttt{:XS}_1\texttt{,YS}_1\texttt{,X}_1\texttt{:ZS}_1\texttt{)} \leftarrow \texttt{app(XS}_1\texttt{,YS}_1\texttt{,ZS}_1\texttt{)}$

σ_0 $\boxed{\texttt{app(XS}_1\texttt{,YS}_1\texttt{,ZS}_1\texttt{)}}$ σ'_0

r_1 $\equiv \texttt{app([],YS}_2\texttt{,YS}_2\texttt{)} \leftarrow$

σ_1 $\square$ σ'_1

Sei $\sigma := \emptyset$.
Wir erhalten für σ_0 und σ_1:

$$\begin{aligned}
\sigma_0 &= mgu(\sigma\,\texttt{app(XS}_0\texttt{,[2,3],[1,2,3])},\texttt{app(X}_1\texttt{: XS}_1\texttt{,YS}_1\texttt{,X}_1\texttt{: ZS}_1\texttt{)})\,|_{\{\texttt{X}_1,\texttt{XS}_1,\texttt{YS}_1,\texttt{ZS}_1\}}\\
&= [\texttt{YS}_1 \mapsto \texttt{[2,3]}, \texttt{X}_1 \mapsto \texttt{1}, \texttt{ZS}_1 \mapsto \texttt{[2,3]}]\\
\sigma_1 &= mgu(\sigma_0\,\texttt{app(XS}_1\texttt{,YS}_1\texttt{,ZS}_1\texttt{)},\texttt{app([],YS}_2\texttt{,YS}_2\texttt{)})\,|_{\{\texttt{YS}_2\}}\\
&= [\texttt{YS}_2 \mapsto \texttt{[2,3]}]
\end{aligned}$$

Warnung: Wir haben jeweils die ‘richtige’ Regel geraten. Ein konkretes Laufzeitsystem muß die Regeln nach einer bestimmten Strategie auswählen (i. a., indem diese von links nach rechts gemäß ihrer textuellen Anordnung im Programm betrachtet werden). Stellt sich diese Wahl (evtl. sehr viel später) als Fehler heraus, dann wird die zuletzt getroffene Auswahl rückgängig gemacht (*Backtracking*).

Formal:
Seien g ein Ziel und σ, σ' idempotente Subst. mit $dom(\sigma) \subseteq Vars$, $dom(\sigma') \subseteq Vars$, sowie $Vars(g) \subseteq Vars$.
Wir schreiben: $\sigma, g \to \sigma'$, falls für eine neue Instanz einer Regel $r \equiv h \leftarrow g_1, \ldots, g_m$ der $mgu(\sigma g, h)$ definiert ist, und wenn für $\sigma_0 = mgu(\sigma g, h)|_{Vars(r)}$ gilt:

$$\begin{aligned}
\sigma_0, g_1 &\to \sigma_1,\\
\sigma_1, g_2 &\to \sigma_2,\\
&\vdots\\
\sigma_{m-1}, g_m &\to \sigma_m
\end{aligned}$$

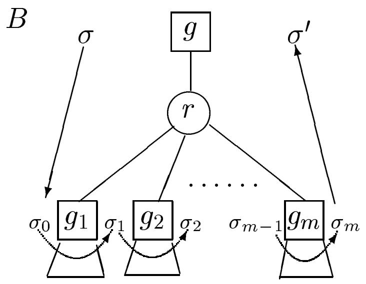
Für σ' gilt dann: $\sigma' = mgu(\sigma g, \sigma_m h) \circ \sigma|_{Vars}$

Dabei ist $Vars(r) = Vars(h) \cup Vars(g_1) \cup \cdots \cup Vars(g_m)$.
Wir bezeichnen σ_0 auch als *Call–to–entry* und σ' als *Exit–to–success*.

Bemerkung: Für den Spezialfall $m = 0$, d. h. die Regel ist ein Fakt, gilt:

$$\begin{aligned}\sigma' &= mgu(\sigma g, \sigma_0 h) \circ \sigma|_{Vars}\\ &= mgu(\sigma g, mgu(\sigma g, h)|_{Vars(r)} h) \circ \sigma|_{Vars}\\ &= mgu(\sigma g, mgu(\sigma g, h) h) \circ \sigma|_{Vars}\\ &= mgu(\sigma g, h) \circ \sigma|_{Vars}\end{aligned}$$

Graphische Darstellung im Baum B:



B heißt *Beweisbaum* für $\langle\sigma, g\rangle$. Notation: $B, \sigma' \vdash \langle\sigma, g\rangle$.

Knüpfen wir an unser Beispiel von oben an, so erhalten wir für die Substitutionen σ' und σ'_0:

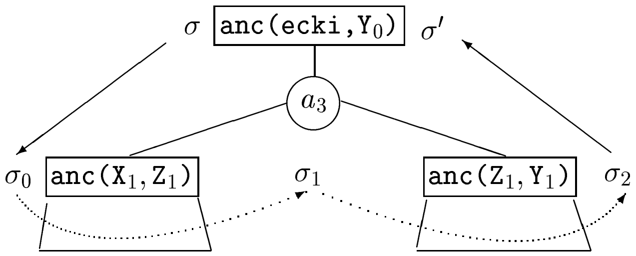
Wegen der Bemerkung von vorhin brauchen wir σ'_1 (und daher auch σ_1) gar nicht zu berechnen und erhalten direkt:

$$\begin{aligned}\sigma'_0 &= mgu(\sigma_0\, \mathtt{app(XS_1,YS_1,ZS_1)}, \mathtt{app([],YS_2,YS_2)}) \circ \sigma_0|_{\{\mathtt{X_1,XS_1,YS_1,ZS_1}\}}\\ &= mgu(\mathtt{app(XS_1,[2,3],[2,3])}, \mathtt{app([],YS_2,YS_2)}) \circ \sigma_0|_{\{\mathtt{X_1,XS_1,YS_1,ZS_1}\}}\\ &= [\mathtt{XS_1} \mapsto \mathtt{[]}, \mathtt{YS_1} \mapsto \mathtt{[2,3]}, \mathtt{X_1} \mapsto 1, \mathtt{ZS_1} \mapsto \mathtt{[2,3]}]\end{aligned}$$

$$\begin{aligned}\sigma' &= mgu(\sigma\, \mathtt{app(XS_0,[2,3],[1,2,3])}, \sigma'_0 \mathtt{app(X_1 : XS_1,YS_1,X_1 : ZS_1)}) \circ \sigma\,|_{\{\mathtt{X_0}\}}\\ &= mgu(\mathtt{app(XS_0,[2,3],[1,2,3])}, \mathtt{app([1],[2,3],[1,2,3])})\,|_{\{X_0\}}\\ &= [\mathtt{X_0} \mapsto \mathtt{[1]}]\end{aligned}$$

Beispiel:

```
    mo(anca,leo)←
    fa(fritz,leo)←
    fa(ecki,anca)←
a1: anc(X,Y)←fa(X,Y)
a2: anc(X,Y)←mo(X,Y)
a3: anc(X,Y)←anc(X,Z),anc(Z,Y)
```



Wir erhalten:

$$\begin{aligned}\sigma_0 &= [\mathtt{X_1} \mapsto \mathtt{ecki}]\\ \sigma_1 &= [\mathtt{X_1} \mapsto \mathtt{ecki}, \mathtt{Z_1} \mapsto \mathtt{anca}]\\ \sigma_2 &= [\mathtt{X_1} \mapsto \mathtt{ecki}, \mathtt{Z_1} \mapsto \mathtt{anca}, \mathtt{Y_1} \mapsto \mathtt{leo}]\\ \sigma' &= [\mathtt{Y_0} \mapsto \mathtt{leo}]\end{aligned}$$

Unter Verwendung des letzten Satzes zeigen wir:

Corollar:
Exit–to–success $\sigma' = mgu(\sigma g, \sigma_m h)) \circ \sigma$ ist idempotent.

Beweis:
Nach Voraussetzung ist σ idempotent. Weiterhin ist der allgemeinste Unifikator zweier Terme idempotent. Laut dem letzten Satz genügt es also zu zeigen:
$im(mgu(\sigma g, \sigma_m h)) \cap dom(\sigma) = \emptyset$
Es gilt: $im(mgu(\sigma g, \sigma_m h)) \subseteq Vars(\sigma g) \cup Vars(\sigma_m h)$
Offenbar ist $Vars(\sigma g) \subseteq im(\sigma) \cup (Vars(g) \setminus dom(\sigma))$
und $Vars(\sigma_m h) \subseteq im(\sigma_m) \cup (Vars(h) \setminus dom(\sigma_m))$.
Da h 'ganz neue' Variablen enthält, genügt es $im(\sigma_m)$ zu betrachten.
$im(\sigma_m) \subseteq im(\sigma_0) \cup \{X \mid X \text{ ist neue Variable}\}$
Deshalb genügt es zu zeigen: $im(\sigma_0) \cap dom(\sigma) = \emptyset$
Denn: $im(\sigma_0) = im(mgu(\sigma g, \sigma_m h)|_{Vars(r)}) \subseteq Vars(\sigma g) \cup Vars(h)$
$\Longrightarrow\ im(\sigma_0) \cap dom(\sigma) \subseteq Vars(\sigma g) \cup dom(\sigma) = \emptyset$ $\square$

Interessante Fragen, die ein Analysator beantworten könnte:

1. Welche Variablen sind *beim Aufruf* eines Ziels g an einen variablenfreien Term gebunden ? Welche sind es *nachher* ? (*Groundness*)
2. Welche Variablen sind *sicher* ungebunden ? (*Freeness*)
3. Welche Variablen sind an denselben Terme gebunden ? (*Sharing*)
4. Gibt es bestimmte Input/Output–Parameter ? (*Mode*)
5. Wie sehen Terme aus, an die eine neue Variable gebunden wird ? (*Type*)
6. Ist die Regelauswahl deterministisch ?

Viele Analysen sind auf präzise Groundness–Information angewiesen (z. B. 6. und 4.)

4.3 Grundlagen der Groundness–Analyse

4.3.1 Konkrete und abstrakte Substitutionen

Sei $Subst_{Vars} := \{\sigma \text{ idempotent} \mid dom(\sigma) \subseteq Vars\} \cup \{FAIL\}$.
Statt konkreten Substitutionen $\sigma \in Subst_{Vars}$, betrachten wir abstrakte Substitutionen $\sigma^{\#}$.
Sei dazu D_{Vars} ein vollständiger Verband abstrakter Substitutionen der Variablen aus *Vars* und $\alpha : Subst_{Vars} \longrightarrow D_{Vars}$ eine (beste) Abstraktionsfunktion mit $\alpha(FAIL) = \bot$ und $\alpha(\sigma_2 \circ \sigma_1) \sqsubseteq \alpha(\sigma_1)$.

Beispiel:

$D_{Vars} := \mathcal{P}(Vars) \cup \{\bot\}$ mit der Ordnung $A \sqsubseteq A'$, sofern $A \supseteq A'$ oder $A = \bot$ für $A, A' \in D_{Vars}$

und $\alpha(\sigma) = \begin{cases} \bot, & \text{falls } \sigma = FAIL \\ \{X \in dom(\sigma) \mid \sigma X \text{ ist variablenfrei}\}, & \text{sonst} \end{cases}$

Beachte: $A \neq \bot$ repräsentiert die Menge aller Substitutionen σ mit $\alpha(\sigma) \supseteq A$ oder $\sigma = FAIL$.

4.3.2 Wie rechnet man mit abstrakten Substitutionen ?

Beispiel: Call–to–Entry ($\sigma_0 = mgu(\sigma g, h)|_{Vars(r)}$)

Ein Term t heiße *ground* genau dann, wenn $Vars(t) = \emptyset$.
Seien $g =$`a(b(X),X)`, $h =$`a(Y,Z)` Terme und $A = \{\mathtt{X}\}$ eine abstrakte Substitution ($A \in D_{Vars}$ aus dem obigen Beispiel).
Dann ist $A\,g =$`a(b(`*ground*`),`*ground*`)` (wir betrachten also die Menge A als eine Substitution $[X \to ground, \forall X \in A]$).
$mgu(A\,g, h) = [\mathtt{Z} \mapsto ground, \mathtt{Y} \mapsto \underbrace{\mathtt{b}(ground)}_{\equiv ground}]$

Dies ist zwar eine mögliche Vorgehensweise, aber formal aufwendig und ineffizient.

Bessere Idee: Forme Call–to–entry und Exit–to–success so um, daß 'möglichst oft' mit konkreten Substitutionen gerechnet wird und 'möglichst selten' Operationen neu zu definieren sind.

$$\text{Call–to–entry:}\quad \sigma_0 \;=\; mgu(\sigma g, h)|_{Vars(r)} \;=\; mgu(mgu(g,h), \sigma) \circ mgu(g,h)|_{Vars(r)}$$

$$\text{Exit–to–success:}\quad \sigma' \;=\; mgu(\sigma g, \sigma_m h) \circ \sigma|_{Vars} \;=\; mgu(mgu(g,h), \sigma + \sigma_m) \circ mgu(g,h)|_{Vars}$$

Bemerkung: Für den Spezialfall $m = 0$, d. h. die Regel h ist ein Faktum, gilt:

$$\sigma' = mgu(\sigma g, h) \circ \sigma|_{Vars} = mgu(mgu(g,h), \sigma) \circ \; mgu(g,h)|_{Vars}$$

'+' in $\sigma + \sigma_m$ bedeutet, daß die beiden unabhängigen Substitutionen σ und σ_m zusammengefügt werden, wobei wir *unabhängig* wie folgt definieren:

Definition:
Zwei Substitutionen σ und σ' heißen *unabhängig*, falls gilt:

$$dom(\sigma) \cap dom(\sigma') = \emptyset \;\wedge\; dom(\sigma) \cap im(\sigma') = \emptyset \;\wedge\; im(\sigma) \cap dom(\sigma') = \emptyset$$

(d. h. $\sigma + \sigma' = \sigma \circ \sigma' = \sigma' \circ \sigma$)

Beispiel: (Fortsetzung)

$g =$`a(b(X),X)` und $h =$`a(Y,Z)`
$mgu(g,h) = [\mathtt{Y} \mapsto \mathtt{b(X)}, \mathtt{Z} \mapsto \mathtt{X}]$
$mgu(mgu(g,h), \{\mathtt{X}\}) \circ mgu(g,h) = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$

4.3.3 Eine Sammlung nützlicher abstrakter Operationen

Addition: $+ : D_{Vars_1} \times D_{Vars_2} \longrightarrow D_{Vars_1 \cup Vars_2}$,
wobei $Vars_1 \cap Vars_2 = \emptyset$ und $\alpha(\sigma_1 + \sigma_2) \sqsubseteq \alpha(\sigma_1) + \alpha(\sigma_2)$.

Spezialisierer: $\Sigma : Subst_{Vars_1} \times D_{Vars_2} \longrightarrow D_{Vars_1 \cup Vars_2}$,
mit $\alpha(mgu(\sigma_1, \sigma_2) \circ \sigma_1) \sqsubseteq \Sigma(\sigma_1, \alpha(\sigma_2))$, sofern $im(\sigma_2) \cap dom(\sigma_1) = \emptyset$.

Restriktion: $(\_)|_{Vars_1} : D_{Vars_2} \longrightarrow D_{Vars_1 \cap Vars_2}$, mit $\alpha(\sigma|_{Vars_1}) \sqsubseteq \alpha(\sigma)|_{Vars_1}$.

Beispiel:

Für $D_{Vars_1} = \mathcal{P}(Vars_1) \cup \{\bot\}$ und $D_{Vars_2} = \mathcal{P}(Vars_2) \cup \{\bot\}$ sind diese Operationen definiert durch ($A_1 \in D_{Vars_1}, A_2 \in D_{Vars_2}$):

- Addition: $A_1 + A_2 := A_1 \cup A_2$
- Restriktion: $A_2|_{Vars_1} := A_2 \cap Vars_1$
- Der Spezialisierer erfülle das Ungleichungssystem:
  a) $\Sigma(\sigma, A_2) \supseteq A_2 \quad (\sigma \in Subst_{Vars_1})$
  b) $\Sigma(\sigma, A_2) \supseteq \{X \mid Vars(\sigma X) \subseteq \Sigma(\sigma, A_2)\}$
  c) $\Sigma(\sigma, A_2) \supseteq Vars(\sigma X)$, sofern $X \in \Sigma(\sigma, A_2)$

Beispielweise gilt für $\sigma = [\mathtt{X} \mapsto \mathtt{Y}, \mathtt{Z} \mapsto 1 : \mathtt{YS}]$ und $A = \{\mathtt{Y}, \mathtt{Z}\}$: $\Sigma(\sigma, A) = \{\mathtt{Y}, \mathtt{Z}, \mathtt{YS}, \mathtt{X}\}$.

Vereinbarung: Wir setzen $A \cup \bot = \bot \cup A = \bot$ und $\bot \cap Vars_1 = \bot$
(Achtung: nicht zu verwechseln mit $A \sqcup \bot = A$ für alle A).

Auflösung des Ungleichungssystems für $\Sigma(\sigma, A)$:

Für eine Variable $X \in dom(\sigma)$ betrachten wir eine korrespondierende Variable $x \in \{0 \sqsubset 1\}$, für die gilt: $x = 1 \iff X \in \Sigma(\sigma, A)$.
Den Variablen aus $\{X_1, \ldots, X_n\} = Vars(\sigma X)$ entsprechen dabei die Variablen $x_1, \ldots, x_n$.

Wir übersetzen das Ungleichungssystem für den Spezialisierer in ein Ungleichungssystem über $x \in \{0 \sqsubset 1\}$:

a) $x \sqsupseteq 1$, für alle $X \in A$

b) $x \sqsupseteq x_1 \sqcap \cdots \sqcap x_n$

c) $x_j \sqsupseteq x$, für $j = 1, \ldots, n$

Von diesem Ungleichungssystem können wir schnell das minimale Modell berechnen. Daraus erhalten wir dann $\Sigma(\sigma, A)$.

Beachte: Wollen wir statt $\Sigma(\sigma, A)$ die Menge $\Sigma(\sigma, A')$ berechnen wollen, muß nur die 1. Zeile des UGS geändert werden. D. h.: Wenn wir den Algorithmus zur Berechnung des kleinsten Modells initialisieren, muß für verschiedene abstrakte Substitutionen A nur jeweils die anfängliche Worklist ausgetauscht werden !

Beispiel:

Sei $\sigma = [\mathtt{X} \mapsto \mathtt{Y} : \mathtt{YS}, \mathtt{Z} \mapsto \mathtt{1} : \mathtt{YS}]$ und $A = \{\mathtt{Y}, \mathtt{Z}\}$.
Wir erhalten das folgende Ungleichungssystem:

$$\begin{array}{lll} x & \sqsupseteq 1 & \text{(a)} \\ y & \sqsupseteq 1 & \text{(b)} \\ x & \sqsupseteq y \sqcap ys & \text{(c)} \\ y & \sqsupseteq x & \text{(d)} \\ ys & \sqsupseteq x & \text{(e)} \\ z & \sqsupseteq ys & \text{(f)} \\ ys & \sqsupseteq z & \text{(g)} \end{array}$$

Die anfängliche Worklist enthält (c) und (g) (dort kommen y und z in den rechten Seiten vor).

4.3.4 Ein 'aussagekräftigerer' Bereich abstrakter Substitutionen: POS

Ziel: Modellierung bedingter Groundness–Info: *Ist* X *ground, dann auch* Y.

Daher definieren wir den Bereich POS zu:

$$\mathtt{POS}_{Vars} := \{\varphi : (Vars \to \{0,1\}) \to \{0,1\} \mid \varphi([X \mapsto 1, \forall X \in Vars]) \equiv \mathbf{1}\} \cup \{\mathbf{0}\}$$

Die Funktion $\varphi = \mathbf{0}$ ($\varphi = \mathbf{1}$) ist dabei diejenige Funktion, die alle Variablenbelegungen auf 0 (auf 1) abbildet.

Die Ordnung auf $\mathtt{POS}_{Vars}$ ist gegeben durch:

$$\varphi_1 \sqsubseteq \varphi_2 :\iff \varphi_1 \to \varphi_2 \equiv \mathbf{1}$$

(d.h. : fr alle $\eta : Vars \to \{0,1\}$ gilt: $\varphi_1(\eta) \sqsubseteq \varphi_2(\eta) :\iff \varphi_1(\eta) \to \varphi_2(\eta) = 1$)

Beispiel:

$\varphi_1 = X_1 \leftrightarrow X_2 \wedge X_3$ und $\varphi_2 = X_1 \to (X_2 \to X_3)$ repräsentieren Funktionen aus $\mathtt{POS}_{\{X_1,X_2,X_3\}} \backslash \{\mathbf{0}\}$. Fr diese beiden Funktionen gilt $\varphi_1 \sqsubseteq \varphi_2$.

Beachte:

- $\varphi \in \mathtt{POS}_{Vars} \backslash \{\mathbf{0}\} \iff \bigwedge_{X \in Vars} X \to \varphi \equiv \mathbf{1}$
  Folglich ist $\mathbf{0}$ das minimale Element in $\mathtt{POS}_{Vars}$. Das nächstgrößere Element ist $\bigwedge_{X \in Vars} X$, das größte $\mathbf{1}$.
- $\varphi_1 \sqcup \varphi_2 = \varphi_1 \vee \varphi_2$
  $\varphi_1 \sqcap \varphi_2 = \varphi_1 \wedge \varphi_2$
- Die Menge A der Variablen, die **sicher** *ground* sind, ist für $\varphi \neq \mathbf{0}$ die Menge $A = \{X \in Vars \mid \varphi \to X \equiv \mathbf{1}\}$.

Beispiel:

$\varphi = X_1 \leftrightarrow X_2 \wedge X_3 \implies A = \emptyset$, d. h.: keine Variable ist sicher *ground*.
$\varphi = (X_1 \to X_2) \wedge X_1 \implies A = \{X_1, X_2\}$, d. h.: beide Variablen sind sicher *ground*,
denn: $(X_1 \to X_2) \wedge X_1 \to X_1 \equiv \mathbf{1} \implies X_1 \in A$
$(X_1 \to X_2) \wedge X_1 \to X_2 \equiv \mathbf{1} \implies X_2 \in A$

Definition:

- $\alpha : Subst_{Vars} \longrightarrow \mathtt{POS}_{Vars}$ ist gegeben durch:

  $$\alpha(FAIL) = \mathbf{0}$$
  $$\alpha(\sigma) = \bigwedge_{X \in dom(\sigma)} (X \leftrightarrow Y_1 \wedge \cdots \wedge Y_k), \text{ wobei } \{Y_1, \ldots, Y_k\} = Vars(\sigma X) \subseteq Vars$$

  Beispiel: Sei $\sigma = [\mathtt{X_1} \mapsto \mathtt{a(X_2,X_3)}, \mathtt{Y} \mapsto \mathtt{b}, \mathtt{Z_1} \mapsto \mathtt{c(Z_2)}]$,
  dann ist $\varphi = \alpha(\sigma) = (X_1 \leftrightarrow X_2 \wedge X_3) \wedge (Y \leftrightarrow 1) \wedge (Z_1 \leftrightarrow Z_2)$
- Addition: $\varphi_1 + \varphi_2 = \varphi_1 \wedge \varphi_2$
- Spezialisierer: $\Sigma(\sigma, \varphi) = \alpha(\sigma) \wedge \varphi$
- Variablenrestriktion: $(\_)|_{Vars_1} : \mathtt{POS}_{Vars_2} \longrightarrow \mathtt{POS}_{Vars_1 \cap Vars_2}$
  Bezeichnen wir die Variablen, die zwar in $Vars_1$, aber nicht in $Vars_2$ liegen, mit Y_i, d.h. $Vars_1 = (Vars_1 \cap Vars_2) \cup \{Y_1, \ldots, Y_k\}$.

Dann definieren wir: $\varphi|_{Vars_1} := \exists Y_1 \cdots \exists Y_k : \varphi = \bigvee_{\substack{(a_1,\ldots,a_k)\\ \in\{0,1\}^k}} \varphi_{Y_1=a_1,\ldots,Y_k=a_k}$

Aus einer Formel φ erhält man $\varphi_{Y=a}$, indem man jedes Vorkommen von Y durch a ersetzt.
Beispielsweise erhlt man: $(X_1 \leftrightarrow X_2 \wedge X_3)_{X_2=0} = X_1 \leftrightarrow 0 = \overline{X}_1 \notin \mathtt{POS}_{\{X_1,X_2,X_3\}}$
$(X_1 \leftrightarrow X_2 \wedge X_3)_{X_2=1} = X_1 \leftrightarrow X_3$

Beispiel:

Mit $\varphi = X_1 \leftrightarrow X_2 \wedge X_3$ erhalten wir fr $\varphi|_{\{X_1,X_3\}} : \mathtt{POS}_{\{X_1,X_2,X_3\}} \longrightarrow \mathtt{POS}_{\{X_1,X_3\}}$:

$$\begin{aligned}
\varphi|_{\{X_1,X_3\}} &= \exists X_2 : (X_1 \leftrightarrow X_2 \wedge X_3)\\
&= (X_1 \leftrightarrow X_2 \wedge X_3)_{X_2=0} \vee (X_1 \leftrightarrow X_2 \wedge X_3)_{X_2=1}\\
&= \overline{X}_1 \vee (X_1 \leftrightarrow X_3) = \cdots = X_1 \rightarrow X_3 \in \mathtt{POS}_{\{X_1,X_3\}}
\end{aligned}$$

4.4 Groundness–Analyse, 1. Versuch

Um nicht zwischen Anfrage ? g und Zielen in Regeln unterscheiden zu müssen, führen wir eine Startregel $s \equiv \mathtt{main} \leftarrow g$ ein und starten mit ? `main`.
Sei *Vars* eine Menge von Variablen. Dann ist $\widetilde{Vars} := \{\tilde{X} \mid X \in Vars\}$.
Weiterhin benötigen wir eine Rename–Funktion ρ, die Variablen aus *Vars* durch entsprechende aus $\widetilde{Vars}$ ersetzt und umgekehrt.
So ein ρ benötigen wir in der Form:

$\rho : Terms_{Vars} \longrightarrow Terms_{\widetilde{Vars}}$ bzw.
$\rho : D_{Vars} \longrightarrow D_{\widetilde{Vars}}$

Idee: Berechne für jedes Ziel g der Regel r' eine abstrakte Substitution $[g] \in D_{Vars(r')}$, so daß für jeden Beweisbaum B mit $B, \emptyset \vdash \langle \emptyset, \mathtt{main}\rangle$ und jeden Teilbaum $B', \sigma' \vdash \langle \sigma, g\rangle$ gilt: $\alpha(\sigma') \sqsubseteq [g]$.
Stelle eine Ungleichungssystems in den Variablen $[r,j]$ $(0 \leq j \leq m)$ auf für eine Regel $r \equiv h \leftarrow g_1, \ldots, g_m$.
Ist dann $g \equiv g_j$, setzen wir $[g] \sqsupseteq [r,j]$.

$[r,0] \sqsupseteq \rho(\Sigma(mgu(g'_j, \rho h), [r', j-1])|_{Vars(\rho r)})$ 'Call–to–entry'
$[r',j] \sqsupseteq \Sigma(mgu(g'_j, \rho h), [r', j-1] + \rho[r,m])|_{Vars(r')}$ 'Exit–to–success',
sofern $r' \equiv h' \leftarrow g'_1, \ldots, g'_k$ und $mgu(g'_j, \rho h) \neq FAIL$.

$[r,j]$ soll als Wert eine abstrakte Substitution erhalten, die alle konkreten Substitutionen repräsentieren soll, die **nach** Beweis des j–ten Ziels in der rechten Seite von r auftreten können.

Spezialfall: m=0
$[r',j] \sqsupseteq \Sigma(mgu(g'_j, \rho h), [r', j-1])|_{Vars(r')}$
(Ist $g \equiv g'_j$, dann $[g] \sqsupseteq [r',j]$)

Beispiel:

0:	$\mathtt{main} \leftarrow \mathtt{a(T,[],1:[])};$	$(h_0 \ \leftarrow \ g_0)$
1:	$\mathtt{a([],Y,Y)} \leftarrow;$	$(h_1 \ \leftarrow)$
2:	$\mathtt{a(X:XS,Y,X:Z)} \leftarrow \mathtt{a(XS,Y,Z)};$	$(h_2 \ \leftarrow \ g_2)$

(? `main` lassen wir nun weg)

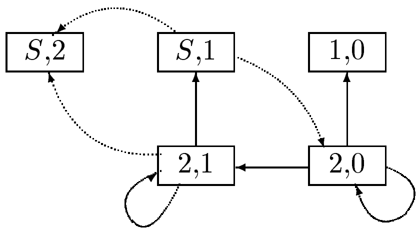
Berechnung der mgu's:

$$
\begin{array}{lclcl}
\sigma_1 & = & mgu(g_0, \rho h_2) & = & [\mathtt{T} \mapsto \mathtt{1}:\widetilde{\mathtt{XS}}, \tilde{\mathtt{X}} \mapsto \mathtt{1}, \tilde{\mathtt{Z}} \mapsto \mathtt{[]}, \tilde{\mathtt{Y}} \mapsto \mathtt{[]}] \\
\sigma_2 & = & mgu(g_2, \rho h_1) & = & [\mathtt{X} \mapsto \mathtt{[]}, \mathtt{Y} \mapsto \tilde{\mathtt{Y}}, \mathtt{Z} \mapsto \tilde{\mathtt{Y}}] \\
\sigma_3 & = & mgu(g_2, \rho h_2) & = & [\mathtt{XS} \mapsto \tilde{\mathtt{X}}:\widetilde{\mathtt{XS}}, \mathtt{Y} \mapsto \tilde{\mathtt{Y}}, \mathtt{Z} \mapsto \tilde{\mathtt{X}}:\tilde{\mathtt{Z}}] \\
\sigma_4 & = & mgu(g_1, \rho h_2) & = & [\tilde{\mathtt{X}} \mapsto \mathtt{1}, \widetilde{\mathtt{XS}} \mapsto \mathtt{S}_1, \mathtt{Y} \mapsto \mathtt{[]}, \mathtt{S} \mapsto \tilde{\mathtt{X}}:\tilde{\mathtt{Z}}]
\end{array}
$$

Ungleichungen:

$[S,0] \sqsupseteq \emptyset$
$[S,1] \sqsupseteq \Sigma(\sigma_1, \rho[2,1]) \cap \{\mathtt{T}\}$
$[1,0] \sqsupseteq \rho(\Sigma(\sigma_2, [2,0]) \cap \{\mathtt{Y}\})$
$[2,0] \sqsupseteq \rho(\Sigma(\sigma_1, \emptyset) \cap \{\tilde{\mathtt{X}}, \widetilde{\mathtt{XS}}, \tilde{\mathtt{Y}}, \tilde{\mathtt{Z}}\})$
$[2,0] \sqsupseteq \rho(\Sigma(\sigma_3, [2,0]) \cap \{\tilde{\mathtt{X}}, \widetilde{\mathtt{XS}}, \tilde{\mathtt{Y}}, \tilde{\mathtt{Z}}\})$
$[2,1] \sqsupseteq \Sigma(\sigma_2, [2,0]) \cap \{\mathtt{X}, \mathtt{XS}, \mathtt{Y}, \mathtt{Z}\}$
$[2,1] \sqsupseteq \Sigma(\sigma_3, [2,0] \cup \rho[2,1]) \cap \{\mathtt{X}, \mathtt{XS}, \mathtt{Y}, \mathtt{Z}\}$

Abhängigkeitsgraph:



Fixpunkt–Iteration:

$[2,0]$	$\{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$
$[1,0]$	$\{\mathtt{Y}\}$
$[2,1]$	$\{\mathtt{X}, \mathtt{XS}, \mathtt{Y}, \mathtt{Z}\}$
$[S,1]$	$\{\mathtt{T}\}$

Ist alle immer prima ?
Nein ! Probleme entstehen bei unterschiedlichen Benutzungen des Prädikats.

Beispiel: Fortsetzung

S: `main` $\leftarrow$ `a(T,[],[1]),a(1:`$\mathtt{S}_1$`,[],`$\mathtt{S}_2$`);` $(h_0 \;\leftarrow\; g_0, g_1)$
(Rest bleibt gleich)

Zusätzliche Ungleichungen (bzw. modifizierte Ungleichungen):
$[S,1] \sqsupseteq \Sigma(\sigma_1, \rho[2,1]) \cap \{\mathtt{T}, \mathtt{S}_1, \mathtt{S}_2\}$
$[S,2] \sqsupseteq \Sigma(\sigma_4, [S,1] \cup \rho[2,1]) \cap \{\mathtt{T}, \mathtt{S}_1, \mathtt{S}_2\}$
$[2,0] \sqsupseteq \rho(\Sigma(\sigma_4, [S,1]) \cap \{\tilde{\mathtt{X}}, \ldots, \tilde{\mathtt{Z}}\})$

Fixpunkt–Iteration: $[2,0] \rightarrow [2,1] \rightarrow [S,1] \rightarrow [2,0]$

	1	2	3	4
$[2,0]$	X,Y,Z	X,Y	Y	Y
$[2,1]$	X,XS,Y,Z	X,XS,Y,Z	XS,Y,Z	Y
$[S,1]$	T	T	T	$\emptyset$

Eigenschaften des 2. Ziels in S zerstören Info über 1. Ziel.

4.5 Groundness–Analyse, 2. Versuch

1. Analyse vermischt Infos über verschiedene Nutzungen von Prädikaten. Die Präzision könnte erhöht werden, wenn wir diese auseinander halten.

Idee: Wir betrachten g als einen ‘Transformator’ (abtsrakter) Substitutionen, d. h. wir betrachten jetzt Paare $\langle A, g\rangle, A \in D_{Vars}$, und untersuchen, welche (abstrakten) Substitution liegt nach Beweis von g vor, wenn vorher A vorlag.

Aufstellen eines UGS:
$\langle A, g\rangle \sqsupseteq \Sigma(mgu(g, \rho h), A + \rho\langle\langle A, g, r\rangle, [r, m]\rangle)|_{Vars(r')}$ (Exit–to–success),
wobei g Ziel ist in rechter Seite von r’ und
$r \equiv h \;\leftarrow\; g_1, \ldots, g_m$ mit $mgu(g, \rho h) \neq FAIL$ und weiterhin
$\langle A, g, r\rangle \sqsupseteq \rho(\Sigma(mgu(g, \rho h), A)|_{Vars(\rho r)})$ (Call–to–entry)
$\langle A, [r, 0]\rangle \sqsupseteq A$
$\langle A, [r, j]\rangle \sqsupseteq \langle\langle A, [r, j-1]\rangle, g_j\rangle$ für $j \geq 1$
Für den speziellen Fall $m = 0$:
$\langle A, g\rangle \sqsupseteq \Sigma(mgu(g, \rho h), A)|_{Vars(r')}$

Beachte:

- $\langle A, g, r\rangle$ ist der alte Call–to–Entry.
- Der Rest der 1. Ungleichung ist Verallgemeinerung des Exit–to–success.
- Die Variablen $\langle A, [r, j]\rangle$ sind Hilfsvariablen mit jeweils nur einer definierten Ungleichung $\Longrightarrow$ könnten eliminiert werden.
- Es gibt immens viele Ungleichungen $\Longrightarrow$ das System kann i. a. überhaupt garnicht repräsentiert werden.
(z. B. enthält $\mathtt{POS}_{Vars}$ $2^{2^n-1} + 1$ viele Elemente !)
- Die Analyse liefert dabei im folgenden Sinne ‘korrekte’ Information:
Gilt: $B, \emptyset \vdash \langle\emptyset, \mathtt{main}\rangle$ und $B, \sigma' \vdash \langle\sigma, g\rangle$, sowie η minimales Modell unseres UGS.
Dann gilt: $\alpha(\sigma') \sqsubseteq \eta(\langle\alpha(\sigma), g\rangle)$

Achtung:

- Die Variablen, von denen der Wert eines Ausdrucks abhängt, werden erst bei der Auswertung überhaupt bestimmt (indirekte Adressierung) !
- Die Ungleichungen tauchen auf für jede abstrakte Substitution A. (sehr viele)

4.6 Fixpunktalgorithmen

Allgemein sind die Ungleichungssysteme, die wir hier betrachten, wie folgt aufgebaut:

Gegeben:

- endlicher vollständiger Verband D
- endliche Menge von 'Transformern' $\mathcal{F}$
- endliche Menge von monotonen Operatoren Ω
- Menge S von Ungleichungsschemata $\langle y, f\rangle \sqsupseteq F$, wobei
  - y ein 'formaler Platzhalter' (Parameter),
  - $f \in \mathcal{F}$ ein Transformer
  - und F ein Ausdruck ist, wobei die Menge der Ausdrücke e definiert ist durch:

$$\begin{array}{rcll} e & ::= & d & \text{(Wert)} \\ & | & y & \text{(Parameter)} \\ & | & op(e_1, \ldots, e_n) & \text{(Operator)} \\ & | & \langle e, f\rangle & \text{(Variable)} \end{array}$$

Das Ungleichungssystem zu S ist $S_D = \{\langle d, f\rangle \sqsupseteq F[d/y] \,|\, \langle y, f\rangle \sqsupseteq F \in S,\, d \in D\}$.

Im Unterschied zu S_D ist S 'sehr klein'. Daher soll ein Lösungsalgorithmus im wesentlichen mit S arbeiten, und vermeiden S_D aufzubauen.

Beispiel:

$D := \{\{a\}, \emptyset\}$

Ungleichungssystem S:

$$\begin{array}{rl} 1: & \langle y, f_1\rangle \sqsupseteq y \cup \langle\langle y, f_2\rangle, f_1\rangle \\ 2: & \langle y, f_2\rangle \sqsupseteq \langle y \cap \{a\}, f_1\rangle \\ 3: & \langle y, f_2\rangle \sqsupseteq \{a\} \cup \langle y, f_1\rangle \end{array}$$

$\Longrightarrow$

Ungleichungssystem S_D:

$$\begin{array}{ll} \langle\emptyset, f_1\rangle & \sqsupseteq \langle\langle\emptyset, f_2\rangle, f_1\rangle \\ \langle\emptyset, f_2\rangle & \sqsupseteq \langle\emptyset, f_1\rangle \\ \langle\emptyset, f_2\rangle & \sqsupseteq \{a\} \cup \langle\emptyset, f_1\rangle \\ \langle\{a\}, f_1\rangle & \sqsupseteq \{a\} \cup \langle\langle\{a\}, f_2\rangle, f_1\rangle \\ \langle\{a\}, f_2\rangle & \sqsupseteq \langle\{a\}, f_1\rangle \\ \langle\{a\}, f_2\rangle & \sqsupseteq \{a\} \cup \langle\{a\}, f_1\rangle \end{array}$$

Um bei der folgenden Behandlung der Fixpunktalgorithmen auf die Ungleichungen aus S zuzugreifen, nummerieren wir sie, wie im Beispiel gezeigt.

Für unser Beispiel erhalten wir so die Mengen: $\{s \mid s \equiv \langle y, f_1\rangle \sqsupseteq F\} = \{1\}$

$\{s \mid s \equiv \langle y, f_2\rangle \sqsupseteq F\} = \{2, 3\}$

4.6.1 Berechnung des 'interessierenden' Teils des kleinsten Modells für S_D

Dazu müssen wir angeben, inwiefern e eine Abbildung auf Variablenbelegungen definiert.
Sei $\mathcal{V}$ die Menge der Variablen von S_D, d. h. $\mathcal{V} = \{\langle d, f\rangle \mid d \in D,\, f \in \mathcal{F}\}$.
Dann ist $[e] : (\mathcal{V} \to D) \to D$ definiert durch

$$\begin{array}{ll} [d](\eta) & = d \\ [op(e_1, \ldots, e_k)](\eta) & = op([e_1](\eta), \ldots, [e_k](\eta)) \;, \\ [\langle e, f\rangle](\eta) & = \eta(\langle [e](\eta), f\rangle) \end{array}$$

wobei η eine Variablenbelegung ist.

Beachte:

- $[e]$ ist monoton für 'monotone' Variablenbelegungen η.
- Eine Variablenbelegung $\eta : \mathcal{V} \longrightarrow D$ heißt monoton, falls $\eta(\langle d, f\rangle) \sqsubseteq \eta(\langle d', f\rangle)$ für alle $d \sqsubseteq d'$.

Seien $\eta \sqsubseteq \eta'$ monoton. Dann gilt: $\eta(\langle [e](\eta), f\rangle) \sqsubseteq \eta'(\langle [e](\eta), f\rangle) \sqsubseteq \eta'(\langle [e](\eta'), f\rangle)$

Für beliebige Variablenbelegungen ist $[e]$ aber keineswegs monoton, d. h. es kann $\eta \sqsubseteq \eta'$ geben mit $[e](\eta) \not\sqsubseteq [e](\eta')$.

Beispiel:

Sei $D = \mathcal{P}(\{X, Y\}) \cup \{\perp\}$ mit der Ordnung $A \sqsubseteq A'$, sofern $A \supseteq A'$ und $\perp \sqsubseteq A'$ für alle $A' \in D$.

Sei $e = \langle\langle \{X\}, f\rangle, f\rangle$ und $\eta :\ \langle \{X\}, f\rangle \mapsto \{X\},\ \langle \emptyset, f\rangle \mapsto \{X, Y\}$
$\eta' :\ \langle \{X\}, f\rangle \mapsto \emptyset,\ \langle \emptyset, f\rangle \mapsto \{X, Y\}$

Dann gilt: $\eta \sqsubseteq \eta'$ und
$[e](\eta) = [\langle\langle \{X\}, f\rangle, f\rangle](\eta) = \eta(\langle \underbrace{[\langle \{X\}, f\rangle](\eta)}_{=\eta(\langle \{X\}, f\rangle)}, f\rangle) = \eta(\langle \{X\}, f\rangle) = \{X\}$,
$[e](\eta') = \eta'(\langle \emptyset, f\rangle) = \{X, Y\}$,
wobei $\{X\} \not\supseteq \{X, Y\}$, und damit $\{X\} \not\sqsubseteq \{X, Y\}$.

Theorem:
S_D besitzt ein kleinstes Modell η. Diese kleinste Modell ist monoton.

Beweis:
(1) Sei η_0 ein Modell von S_D, d. h. $\eta_0(\langle d, f\rangle) \sqsupseteq [F[d/y]](\eta_0)$ für alle $\langle d, f\rangle \sqsupseteq F[d/y] \in S_D$.

Wir zeigen: Es exisitiert ein monotones Modell η_1 von S_D mit $\eta_1 \sqsubseteq \eta_0$.

Beweis: $\eta_1(\langle d, f\rangle) := \bigsqcap_{d \sqsubseteq d'} \eta_0(\langle d', f\rangle)$
Offenbar ist η_1 monoton und $\eta_1 \sqsubseteq \eta_0$.

Bleibt zu zeigen, daß η_1 Modell von S_D ist.
Betrachte $\langle d, f\rangle \sqsupseteq F[d/y] \in S_D$.
Es genügt zu zeigen: $\eta_0(\langle d', f\rangle) \sqsubseteq [F[d/y]](\eta_1)$ für alle $d \sqsupseteq d'$ $\quad (*)$
Da $\eta_0(\langle d', f\rangle) \sqsupseteq [F[d'/y]](\eta_0)$, folgt $(*)$ aus $[F[d'/y]](\eta_0) \sqsupseteq [F[d/y]](\eta_1) \quad (**)$

$(**)$ beweisen wir durch Induktion über die Struktur von F:
Sei e ein Teilausdruck von F.

$e \equiv y$: $[e[d'/y]](\eta_0) = d' \sqsupseteq d = [e[d/y]](\eta_1)$
$e \equiv d_1$: $[e[d'/y]](\eta_0) = d_1 = [e[d/y]](\eta_1)$
$e \equiv op(e_1, \ldots e_n)$, dann folgt $[e[d'/y]](\eta_0) \sqsupseteq [e[d'/y]](\eta_1)$ aus der Monotonie von op und der Ind.vor. für e_i.
$e \equiv \langle e', f\rangle$: Es gilt nach Ind.vor. $[e'[d'/y]](\eta_0) \sqsupseteq [e'[d/y]](\eta_1)$
Daraus folgern wir:

$$\begin{array}{rl} & \eta_0(\langle[e'[d'/y]](\eta_0), f\rangle)\ , \\ \sqsupseteq & \eta_1(\langle[e'[d'/y]](\eta_0), f\rangle) \\ \sqsupseteq & \eta_1(\langle[e'[d/y]](\eta_1), f\rangle) \end{array}$$

wegen der Monotonie von η_1 und der Ind.vor.

(2) Unter den monotonen Modellen gibt es ein kleinstes Modell (folgt aus unserer bisherigen Überlegung).

(3) Sei η das kleinste Modell und η' ein nicht–monotones Modell.
Zu zeigen: $\eta \sqsubseteq \eta'$
(1) $\Longrightarrow \exists \eta_1$ monoton: η_1 Modell und $\eta_1 \sqsubseteq \eta'$
(2) $\Longrightarrow \eta \sqsubseteq \eta_1$

Daraus folgt: $\eta \sqsubseteq \eta'$, d. h. η ist das kleinste Modell überhaupt ! □

Vermeidung der Konstruktion von S_D

Idee:

- Konstruiere von S_D nur soviel, wie nötig ist, um Werte der uns interessierenden Variablen zu berechnen. Insbesondere berechnen wir nur eine partielle Variablenbelegung η.
- Berechne die Variablenabhängigkeiten **dynamisch** !

Im Folgenden wollen wir zwei Algorithmen (Solver) zur Lösung von S_D für eine uns interessierende Variablenmenge X angeben.

4.6.2 Der Worklist–Solver W

Sei X die Menge der interessierenden Variablen $\langle d, f\rangle \in \mathcal{V}$.

Algorithmus:

```
{
    ⋮
  /* Initialisierung */
  W := ∅;
  forall (⟨d,f⟩ ∈ V) {
    W := W ∪ {(d,s) | s ≡ ⟨y,f⟩ ⊒ F ∈ S};
    η(⟨d,f⟩) := ⊥;     /* füge ⟨d,f⟩ zu dom(η) hinzu */
    Infl(⟨d,f⟩) := ∅;    /* Infl(x) ist Menge der Ungleichungen, die x benutzen */
    }
  while (W ! = ∅)
    Solve(Extract(W)); /* erstes Element aus W entfernen */
      ⋮
}
```

```
/* Solve wertet Ungleichungen aus */
void Solve(D d, Inequality s ≡ ⟨y,f⟩ ⊒ F) {
D new;
  new := Eval(d,F,s) ⊔ η(⟨d,f⟩);
  if (η(⟨d,f⟩) != new) {    /* neue Approximation gefunden ! */
    η(⟨d,f⟩) := new;
    W := W ∪ Infl(⟨d,f⟩);    /* alle Ungl., die alten Wert benutzen, merken */
    Infl(⟨d,f⟩) := ∅;
    }
}

/* Eval wertet Teilausdruck e von F aus */
void Eval(D d, Exp e, Inequality s ≡ ⟨y,f⟩ ⊒ F) {
D value;
  switch (e) {
    case d′: return d′
    case y: return d;
    case op(e_1,...,e_k):
      return op(Eval(d,e_1,s),...,Eval(d,e_k,s));
    case ⟨e′,f′⟩:
      value := Eval(d,e′,s);
      if (⟨value,f′⟩ ∉ dom(η)) {    /* neue Variable gefunden */
        η(⟨value,f′⟩) := ⊥;
        Infl(⟨value,f′⟩) := ∅;
        forall (s′ ≡ ⟨y,f′⟩ ⊒ F ∈ S)
          W := W ∪ {(value,s′)};
        }
      /* initiiere Berechnung von η(⟨value,f⟩) */
      Infl(⟨value,f′⟩) := Infl(⟨value,f′⟩) ∪ {(d,s)};
      /* bei der Auswertung von (d,s) wurde auf den */
      /*    Wert von ⟨value,f′⟩ zugegriffen */
      return η(⟨value,f′⟩);
    }  /* switch */
}
```

Korrektheit

1. Für jede Variable $\langle d,f\rangle$ in $dom(\eta)$ bilden die bezeichneten Werte $\eta(\langle d,f\rangle)$ eine **echt** aufsteigende Kette. $\Longrightarrow$ Der Algorithmus terminiert.

2. Sei $\bar{\eta}$ kleinstes Modell. Dann gilt stets: $\eta(\langle d,f\rangle) \sqsubseteq \bar{\eta}(\langle d,f\rangle)$.

3. Ist schließlich $W = \emptyset$, dann gilt: $\eta(\langle d,f\rangle) \sqsupseteq [F[d/y]](\eta)$ für alle $d \in D, \langle y,f\rangle \sqsupseteq F \in S$ mit $\langle d,f\rangle \in dom(\eta)$.

Zum Beweis zeigt man induktiv, daß nach jedem Aufruf von **Solve**() gilt:
W enthält alle Paare (d,s) mit $s \equiv \langle y,f\rangle \sqsupseteq F \in S$, so daß (a) und (b) gelten:

(a) $\eta(\langle d,f\rangle) \not\sqsupseteq [F[d/y]](\eta)$

(b) Es gibt eine Folge von Paaren $(d_0,s_0),\ldots,(d_k,s_k)$ mit $s_j \equiv \langle y,f_j\rangle \sqsupseteq F_j \in S$, so daß

- $\langle d_j,f_0\rangle \in V$;

- $(d_k, s_k) = (d, s)$, sowie
- für $j = 0, \ldots, k-1$ gilt:
  bei der letzten Auswertung von $F_j[d_j/y]$, bei der $\eta(\langle d_j, f_j\rangle)$ modifiziert wurde, wurde auf $\eta(\langle d_{j+1}, f_{j+1}\rangle)$ zugegriffen.

4. Sei η die berechnete partielle Abbildung. Dann ist $\eta = \bar{\eta}|_{dom(\eta)}$. Beweisidee: Füge alle weiteren Paare (d, r) zu $Vars \in \mathcal{V} \setminus dom(\eta)$ in die anfängliche Worklist ein.
Benutze zur Extraktion aus W die Strategie, daß zuerst nur Paare extrahiert werden, die der Algorithmus **ohne** Auffüllen von W entnommen hatte. Sei $\tilde{\eta}$ die dann berechntete Abbildung. Für $\tilde{\eta}$ gilt:

- $\eta = \tilde{\eta}|_{dom(\eta)}$
- $\tilde{\eta} \sqsubseteq \bar{\eta}$ (wegen 2.)
- $\tilde{\eta}$ ist ein Modell (wegen 3.) $\implies$ $\tilde{\eta} = \bar{\eta}$

Zur Laufzeit: (bei unserer Anwendung)

- Für jedes $f \in \mathcal{F}$ gibt es i. a. nur wenige Ungleichungsschemata $\langle y, f\rangle \sqsupseteq F \in S$.
- Pro erzeugter Variable $\langle d, f\rangle$ werden i. a. nur wenige Iterationen furchgeführt (≤ 5).
- Für jedes $f \in \mathcal{F}$ ist die Menge $\{\langle d, f\rangle \mid \langle d, f\rangle \in dom(\eta)\}$ sehr klein ($\sharp\{\ldots\} \leq 5$).

$\implies$ Die Laufzeit ist ‘praktisch’ linear in $\sharp S$ ($\approx$ Programmgröße).

Verbesserungen

- Für unsere Anwendung ist stets $\eta(\langle d, f\rangle) \sqsupseteq d$.
  D. h. wenn wir auf eine Variable $\langle d, f\rangle$ stoßen, so können wir statt $\perp$ mit d initialisieren.
- Statt beim Auffinden einer neuen Variablen $\langle d, f\rangle \notin dom(\eta)$ die zuständigen (d, s) in W einzutragen, berechnen wir erst einmal einen anständigen Anfangswert für $\eta(\langle d, f\rangle)$, indem wir die rechten Seiten aller zuständigen Ungleichungen erst einmal auswerten.
- Technisch (Fecht ’95):
  Ersetzte den Programmteil `forall` $(s' \equiv \langle y, f'\rangle \sqsupseteq F \in S)$
  $W := W \cup \{(value, s')\}$;
  in **Eval**() durch das Fragment `forall` $(s' \equiv \langle y, f'\rangle \sqsupseteq F \in S)$.
  Solve$(value, s')$;

4.6.3 Der Topdown–Solver TD

Idee: Bevor wir auf den Wert $\eta(\langle d, f\rangle)$ zugreifen, versuchen wir eine möglichst gute Approximation an ihre letztendliche Lösung zu berechnen.
Dazu modifizieren wir **Solve**() derart, daß es direkt eine Fixpunktiteration ausführt.
Für jedes Paar (d, r) mit $r \equiv \langle y, f\rangle \sqsupseteq F$ verwalten wir Werte $called(d, r)$ und $stable(d, r)$, wobei gilt:

$called(d, r) == 1$	$\iff$	**Solve**(d, r) wurde bereits aufgerufen, aber noch nicht beendet.
$stable(d, r) == 1$	$\iff$	**Solve**(d, r) wurde bereits beendet und der ‘Wert’ ist gültig.

Sei X wiederum die Menge der interessierenden Variablen.

Algorithmus: (VanHentenryck,LeCharlier '92)

```
{
    ⋮
  /* Hauptprogramm */
  forall (⟨d,f⟩ ∈ X) {
    if (⟨value,f'⟩ ∉ dom(η)) {
      η(⟨d,f⟩) := ⊥;     /* Initialisierung */
      Infl(⟨d,f⟩) := ∅ ;
      forall (r ≡ ⟨y,f⟩ ⊒ F) {
        stable(d,r) := 0;
        called(d,r) := 0;
        Solve(d,r);
        }
    } else {
      forall (r ≡ ⟨y,f⟩ ⊒ F)
        Solve(d,r);
      }  /* if */
    }
    ⋮
}

/* Solve wertet Ungleichungen aus */
void Solve(D d, Inequality r ≡ ⟨y,f⟩ ⊒ F) {
D new;
  if (!called(d,r)) {    /* ist η(⟨d,r⟩) bereits in Bearbeitung, brauchen */
                         /*   wir keine weitere Beobachtung zu starten */
    called(d,r) := 1;
    while (!stable(d,r)) {    /* Fixpunkt–Iteration */
      stable(d,r) := 1;
      new := Eval(d,F,r) ⊔ η(⟨d,f⟩);
      if (η(⟨d,f⟩) != new) {
        η(⟨d,f⟩) := new;
        Infl(⟨d,f⟩) := TransClosure(Infl(⟨d,f⟩));     (*)
        forall ((d',r') ∈ Infl(⟨d,f⟩)) {
          stable(d',r') := 0;
          Infl(⟨d',f'⟩) := ∅;     /* wobei r' ≡ ⟨y,f'⟩ ⊒ F' */
          }
        Infl(⟨d,f⟩) := ∅;
        }   /* if */
      }   /* while */
      called(d,r) := 0;
    }   /* if */
}
```

(∗) **TransClosure**(M) berechnet die transitive Hülle der Variablenabhängigkeiten.
Die transitive Hülle der Menge M ist dabei die kleinste Menge M', die sowohl M enthält, als

auch für jedes Element $x \in M'$ die Menge $\mathit{Infl}(x)$.

```
/* Eval wertet Teilausdruck e von F aus */
void Eval(D d, Exp e, Inequality r ≡ ⟨y,f⟩ ⊒ F) {
D value;
  switch (e) {
    case d': return d'
    case y: return d;
    case op(e1,...,ek): return op(Eval(d,e1,r),...,Eval(d,ek,r));
    case ⟨e',f'⟩:
      value := Eval(d,e',r);
      if (⟨value,f'⟩ ∉ dom(η)) {    /* neue Variable gefunden */
        η(⟨value,f'⟩) := ⊥;
        Infl(⟨value,f'⟩) := ∅;
        forall (r' ≡ ⟨y,f'⟩ ⊒ F) {
          stable(value,r') := 0;
          called(value,r') := 0;
          }
        }   /* if */
      forall (r' ≡ ⟨y,f'⟩ ⊒ F')
        Solve(value,r');
      Infl(⟨value,f'⟩) := Infl(⟨value,f'⟩) ∪ {(d,r)};
      return η(⟨value,f'⟩);
    }   /* switch */
}
```

Bemerkung:
Der Topdown–Iterierer ist besser als der Worklist–Algorithmus für sehr kleine Ungleichungssysteme, bzw. für solche, wo der Abhängigkeitsgraph zwischen den ('interessierenden') Variablen von S_D kleine starke Zusammenhangskomponenten hat.
Für große Ungleichungssysteme und große starke Zusammenhangskomponenten eignet sich der Worklist–Algorithmus besser !

Zur Implementierung der Algorithmen:

- η:
  (1) benutze ein Feld E mit Indizierung $\mathcal{F}$;
  jedes $E[f]$ ($f \in \mathcal{F}$) enthält eine Liste von Paaren $(d, \eta(\langle d, f\rangle))$
  (2) Hash–Tabelle

- *stable*, *called*:
  (1) benutze ein Feld SC indiziert mit Ungleichungsschemata aus S;
  d. h. jedes $SC[r]$ (wobei $r \equiv \langle y, f\rangle \sqsupseteq F$) enthält eine Liste von Tripeln $(d, \mathit{stable}(d,r), \mathit{called}(d,r))$
  (2) Hash–Tabelle

Beispiel:

$D := \mathcal{P}(\{a,b\}),\ \perp := \emptyset$

Ungleichungssystem S:

$$\begin{array}{rl}
1: & \langle y, f_1\rangle \sqsupseteq (y \cap \{b\}); \{a\} \\
2: & \langle y, f_2\rangle \sqsupseteq \langle\langle y, f_2\rangle \cup \{b\}, f_3\rangle \\
3: & \langle y, f_3\rangle \sqsupseteq \langle y, f_1\rangle \\
4: & \langle y, f_3\rangle \sqsupseteq \langle y, f_2\rangle \cup y
\end{array}$$

Gesucht: $\langle\emptyset, f_3\rangle$, d.h.: $X = \{\langle\emptyset, f_3\rangle\}$

Lösung:

W	*called*	η	*Infl*
$\langle\emptyset,4\rangle, \langle\emptyset,3\rangle$	$-$	$\langle\emptyset, f_3\rangle = \emptyset$	$\langle\emptyset, f_3\rangle : -$
$\langle\emptyset,3\rangle$	$\langle\emptyset,4\rangle$		
	$\langle\emptyset,2\rangle, \langle\emptyset,4\rangle$	$\langle\emptyset, f_2\rangle = \emptyset$	$\langle\emptyset, f_2\rangle : -$
			$\langle\emptyset, f_2\rangle : -\langle\emptyset,2\rangle$
	$\langle\{b\},3\rangle, \langle\emptyset,2\rangle, \langle\emptyset,4\rangle$	$\langle\{b\}, f_3\rangle = \emptyset$	$\langle\{b\}, f_3\rangle : -$
	$\langle\{b\},1\rangle, \langle\{b\},3\rangle, \langle\emptyset,2\rangle, \langle\emptyset,4\rangle$	$\langle\{b\}, f_1\rangle = \emptyset$	$\langle\{b\}, f_1\rangle : -\langle\{b\},3\rangle$
		$\langle\{b\}, f_1\rangle = \{a\}$	
	$\langle\{b\},3\rangle, \langle\emptyset,2\rangle, \langle\emptyset,4\rangle$	$\langle\{b\}, f_3\rangle = \{a\}$	
	$\langle\{b\},4\rangle, \langle\emptyset,2\rangle, \langle\emptyset,4\rangle$		
	$\langle\{b\},2\rangle, \langle\{b\},4\rangle, \langle\emptyset,2\rangle, \langle\emptyset,4\rangle$	$\langle\{b\}, f_2\rangle = \emptyset$	$\langle\{b\}, f_2\rangle : -\langle\{b\},2\rangle$
$\langle\{b\},2\rangle, \langle\emptyset,3\rangle$	$\langle\{b\},4\rangle, \langle\emptyset,2\rangle, \langle\emptyset,4\rangle$	$\langle\{b\}, f_2\rangle = \{a\}$	$\langle\{b\}, f_2\rangle : -$
	$\langle\emptyset,2\rangle, \langle\emptyset,4\rangle$	$\langle\{b\}, f_3\rangle = \{a,b\}$	$\langle\{b\}, f_2\rangle : -\langle\{b\},4\rangle$
			$\langle\emptyset, f_2\rangle : -\langle\emptyset,2\rangle$
	$\langle\emptyset,4\rangle$	$\langle\emptyset, f_2\rangle = \{a,b\}$	$\langle\{b\}, f_3\rangle : -\langle\emptyset,2\rangle$
$\langle\emptyset,2\rangle, \ldots, \ldots$			$\langle\emptyset, f_2\rangle : -$
		$\langle\emptyset, f_3\rangle = \{a,b\}$	$\langle\emptyset, f_2\rangle : -\langle\emptyset,4\rangle\langle\emptyset,2\rangle$
$\langle\{b\},2\rangle, \langle\emptyset,3\rangle$	$\langle\emptyset,2\rangle$		$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Ergebnis:

$$\begin{array}{ll}
\langle\emptyset, f_3\rangle & = \{a,b\} \\
\langle\emptyset, f_2\rangle & = \{a,b\} \\
\langle\{b\}, f_3\rangle & = \{a,b\} \\
\langle\{b\}, f_1\rangle & = \{a\} \\
\langle\{b\}, f_2\rangle & = \{a,b\} \\
\langle\{a,b\}, f_3\rangle & = \{a,b\} \\
\langle\{a,b\}, f_1\rangle & = \{a\} \\
\langle\{a,b\}, f_2\rangle & = \{a,b\} \\
\langle\emptyset, f_2\rangle & = \emptyset
\end{array}$$