# 5 Optimierung logischer Programme

Wir betrachten hier nur die Mini-Sprache PuP (“Pure Prolog”).
Insbesondere verzichten wir (erst einmal) auf:

- Arithmetik;
- den Cut-Operator.
- Selbst-Modifikation von Programmen mittels assert und retract.

## Beispiel:

$$
\begin{array}{lcl}
\text{bigger}(X,Y) & \leftarrow & X = \textit{elephant}, Y = \textit{horse} \\
\text{bigger}(X,Y) & \leftarrow & X = \textit{horse}, Y = \textit{donkey} \\
\text{bigger}(X,Y) & \leftarrow & X = \textit{donkey}, Y = \textit{dog} \\
\text{bigger}(X,Y) & \leftarrow & X = \textit{donkey}, Y = \textit{monkey} \\
\text{is_bigger}(X,Y) & \leftarrow & \text{bigger}(X,Y) \\
\text{is_bigger}(X,Y) & \leftarrow & \text{bigger}(X,Z), \text{is_bigger}(Z,Y) \\
? \quad \text{is_bigger}(\textit{elephant}, \textit{dog}) & &
\end{array}
$$

## Ein realistischeres Beispiel:

$$\begin{array}{lll} \mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\; Y = Z \\ \mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z') \\ ? \quad \mathsf{app}(X, [Y, c], [a, b, Z]) & & \end{array}$$

## Ein realistischeres Beispiel:

$$\mathsf{app}(X, Y, Z) \quad \leftarrow \quad X = [\,], \; Y = Z$$

$$\mathsf{app}(X, Y, Z) \quad \leftarrow \quad X = [H|X'], \; Z = [H|Z'], \; \mathsf{app}(X', Y, Z')$$

$$? \quad \mathsf{app}(X, [Y, c], [a, b, Z])$$

## Bemerkung:

| | | |
|---|---|---|
| $[\,]$ | $==$ | das Atom leere Liste |
| $[H\|Z]$ | $==$ | binäre Konstructor-Anwendung |
| $[a, b, Z]$ | $==$ | Abkürzung für: $[a\|[b\|[Z\|[\,]]]]$ |

Ein Programm $p$ ist darum wie folgt aufgebaut:

$$
\begin{array}{lcl}
t & ::= & a \mid X \mid _ \mid f(t_1, \ldots, t_n) \\
g & ::= & p(t_1, \ldots, t_k) \mid X = t \\
c & ::= & p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\
p & ::= & c_1 \ldots . c_m ? g
\end{array}
$$

- Ein Term $t$ ist entweder ein Atom, eine (evt. anonyme) Variable oder eine Konstruktor-Anwendung.
- Ein Ziel $g$ ist entweder ein Literal, d.h. ein Prädikats-Aufruf, oder eine Unifikation.
- Eine Klausel $c$ besteht aus einem Kopf $p(X_1, \ldots, X_k)$ sowie einer Folge von Zielen als Rumpf.
- Ein Programm besteht aus einer Folge von Klauseln sowie einem Ziel als Anfrage.

## Prozedurale Sicht auf PuP-Programme:

| | | |
|---|---|---|
| Ziel | $=$ | Prozedur-Aufruf |
| Prädikat | $=$ | Prozedur |
| Definition | $=$ | Rumpf |
| Term | $=$ | Wert |
| Unifikation | $=$ | elementarer Berechnungschritt |
| Bindung von Variablen | $=$ | Seiteneffekt |

## Achtung: Prädikat-Aufrufe ...

- liefern keinen Rückgabewert!
- beeinflussen den Aufrufer einzig durch Seiteneffekte :-)
- können fehlschlagen. Dann wird die nächste Definition probiert $\Longrightarrow$ backtracking

## Ineffizienzen:

**Backtracking:**

- Die passende Alternative muss gefunden werden $\Longrightarrow$ Indexing
- Weil ein erfolgreicher Aufruf später noch in eine Sackgasse führen kann, kann bei weiteren offenen Alternativen der Keller nicht geräumt werden.

**Unifikation:**

- Die Übersetzung muss gegebenenfalls zwischen Überprüfung Aufbau hin und her schalten.
- Bei Unifikation mit Variable muss ein Occur Check durchgeführt werden.

**Typüberprüfung:**

- Weil Prolog ungetypt ist, wird oft erst zur Laufzeit sicher gestellt, dass ein Term von der gewünschten Form ist.
- Andernfalls könnte es unangenehme Fehler geben.

## Einige Optimierungen:

- Umwandlung letzter Aufrufe in Sprünge;
- Compilezeit-Typinferenz;
- Identifizierung der Determiniertheit von Prädikaten ...

## Beispiel:

$$\mathsf{app}(X, Y, Z) \quad \leftarrow \quad X = [\,],\; Y = Z$$
$$\mathsf{app}(X, Y, Z) \quad \leftarrow \quad X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z')$$
$$?\quad \mathsf{app}([a, b], [Y, c], Z)$$

## Beobachtung:

- In PuP müssen Funktionen durch Prädikate simuliert werden.
- Diese verfügen dann über Input- und Output-Parameter.
- Als Input-Parameter bezeichnen wir solche, die bei jedem Aufruf mit einem variablen-freien Term instantiiert sind.

  Diese heißen auch ground.
- Im Beispiel ist der erste Parameter von app ein Input-Parameter.
- Unifikation mit diesem Parameter kann als Pattern Matching implementiert werden !
- Dann zeigt sich, dass app deterministisch ist !!!

## 5.1 Groundness-Analyse

Eine Variable $X$ heißt ground bzgl. einer Programmausführung $\pi$ vom Startpunkt des Programms zu einem Programmpunkt $v$, falls $X$ an einen variablenfreien Term gebunden ist.

Ziel:

- Finde die Variablen, die bei Erreichen eines Programmpunkts ground sind !
- Find die Argumente eines Prädikats, die bei jedem Aufruf ground sind !

## Idee:

- Beschreibe Groundness durch Werte aus $\mathbb{B}$:

  $1$ $=$ definitiv variablenfreier Term;

  $0$ $=$ Term, der definitiv Variablen enthält.

- Eine Menge von Variablenbelegungen beschreiben wir durch Boolesche Funktionen :-)

  $X \leftrightarrow Y$ $=$ $X$ ist genau dann ground wenn $Y$ ground ist.

  $X \wedge Y$ $=$ $X$ und $Y$ sind ground.

## Idee (Forts.):

- Die konstante Funktion $0$ bezeichnet einen unerreichbaren Programmpunkt.
- Vorkommende Mengen von Variablenbelegungen sind unter Substitution abgeschlossen.

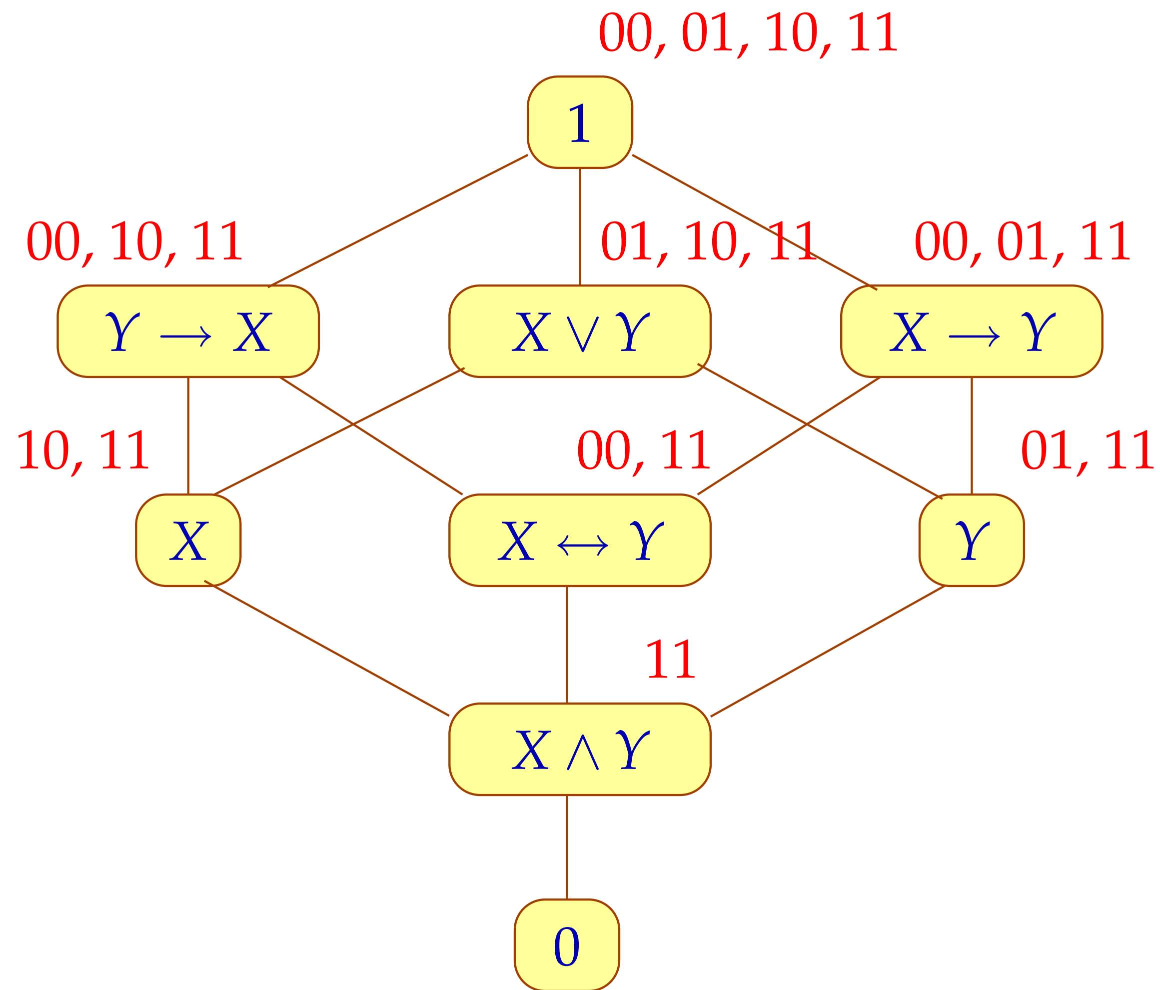
  Das heißt für jede vorkommende Funktion $\phi \neq 0$,

  $$\phi(1, \ldots, 1) = 1$$

  Diese Funktionen heißen positiv.
- Die Menge aller dieser Funktionen nennen wir $\mathsf{Pos}$.

  Ordnung: $\phi_1 \sqsubseteq \phi_2$ falls $\phi_1 \Rightarrow \phi_2$.
- Insbesondere ist $0$ das kleinste Element :-)

## Beispiel:



# Bemerkungen:

- Nicht alle positiven Funktionen sind monoton !!!
- Bei $k$ Variablen gibt es $2^{2^k-1}+1$ viele Funktionen.
- Die Höhe dieses Verbands ist $2^k$.
- Wir konstruieren eine interprozedurale Analyse, die für jedes Prädikat $p$ eine (monotone) Transformation

$$[\![p]\!]^\sharp : \mathsf{Pos} \to \mathsf{Pos}$$

berechnet.

- Für jede Regel $p(X_1,\ldots,X_k) \Leftarrow g_1,\ldots,g_n$ haben wir die Ungleichungen:

$$[\![p]\!]^\sharp\,\psi \quad \sqsupseteq \quad \exists\, X_{k+1},\ldots,X_m.\; [\![g_n]\!]^\sharp\,(\ldots([\![g_1]\!]^\sharp\,\psi)\ldots)$$

// $m$ Anzahl der Variablen der Klausel

## Abstrakte Unifikation:

$$\llbracket X = t \rrbracket^\sharp \psi \quad = \quad \psi \wedge (X \leftrightarrow X_1 \wedge \ldots \wedge X_r)$$

$$\text{falls} \quad \mathit{Vars}(t) = \{X_1, \ldots, X_r\}.$$

## Abstraktes Literal:

$$\llbracket q(s_1, \ldots, s_k) \rrbracket^\sharp \psi \quad = \quad \mathsf{combine}^\sharp_{s_1,\ldots,s_k}(\psi, \llbracket q \rrbracket^\sharp (\mathsf{enter}^\sharp_{s_1,\ldots,s_k} \psi))$$

// analog einem Prozeduraufruf !!

## Dabei ist:

$$\mathsf{enter}^\sharp_{s_1,\ldots,s_k}\psi \;=\; \mathsf{ren}\,(\exists\, X_1,\ldots,X_m.\; [\![\bar{X}_1 = s_1,\ldots,\bar{X}_k = s_k]\!]^\sharp\psi)$$

$$\mathsf{combine}^\sharp_{s_1,\ldots,s_k}(\psi,\psi_1) \;=\; \exists\, \bar{X}_1,\ldots,\bar{X}_r.\; \psi \wedge [\![\bar{X}_1 = s_1,\ldots,\bar{X}_k = s_k]\!]^\sharp(\overline{\mathsf{ren}}\,\psi_1)$$

wobei

$$\exists\, X.\,\phi \;=\; \phi[0/X] \vee \phi[1/X]$$

$$\mathsf{ren}\,\phi \;=\; \phi[X_1/\bar{X}_1,\ldots,X_k/\bar{X}_k]$$

$$\overline{\mathsf{ren}}\,\phi \;=\; \phi[\bar{X}_1/X_1,\ldots,\bar{X}_r/X_r]$$