

Diese operationelle Semantik ist einigermaßen **realistisch** :-)

## Kosten eines Prozedur-Aufrufs:

**Vor Betreten des Rumpfs:** ● Anlegen eines Kellerrahmens;

- Retten der Register;
- Retten der Fortsetzungsadresse;
- Anspringen des Rumpfs.

**Bei Beenden des Aufrufs:** ● Aufgeben des Kellerrahmens;

- Restaurieren der Register;
- Übergeben des Ergebnisses;
- Rücksprung hinter die Aufrufstelle.

⇒ ... ziemlich teuer !!!

## 1. Idee: Inlining

Kopiere den Funktionsrumpf an jede Aufrufstelle !!!

Beispiel:

```
abs (a1) {  
    b1 = a1;  
    b2 = -a1;  
    ret = max (b1, b2);  
    return ret;  
}  
  
max (a1, a2) {  
    if (a1 < a2) return a2;  
    return a1;  
}
```

... liefert:

```
abs (a1) {  
    b1 = a1;  
    b2 = -a1;  
    a1 = b1;  
    a2 = b2;  
    if (a1 < a2) { ret = a2; goto _max; }  
    ret = a1; goto _max;  
_max : return ret;  
}
```

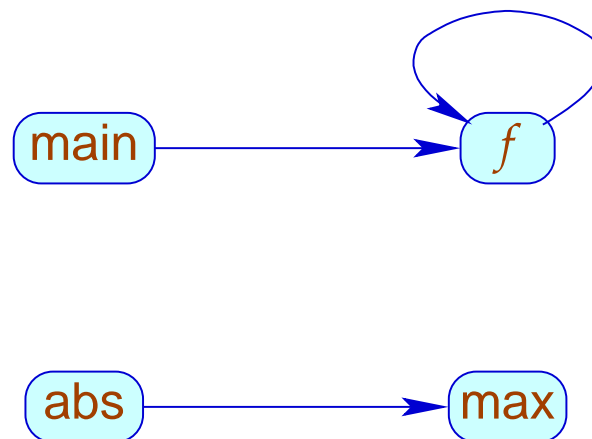
## Probleme:

- Der einkopierte Block modifiziert evt. die  $a_i, b_i$  :-(  
• Allgemeiner: Mehrfachbenutzung gleicher Variablennamen kann zu Fehlern führen;  
• Mehrfach-Verwendung einer Funktion führt zu Code-Duplizierung :-((  
• Wie gehen wir mit **Rekursion** um ???

## Erkennen von Rekursion:

Wir konstruieren den **Aufruf-Graph** des Programms.

In den Beispielen:



## Aufruf-Graph:

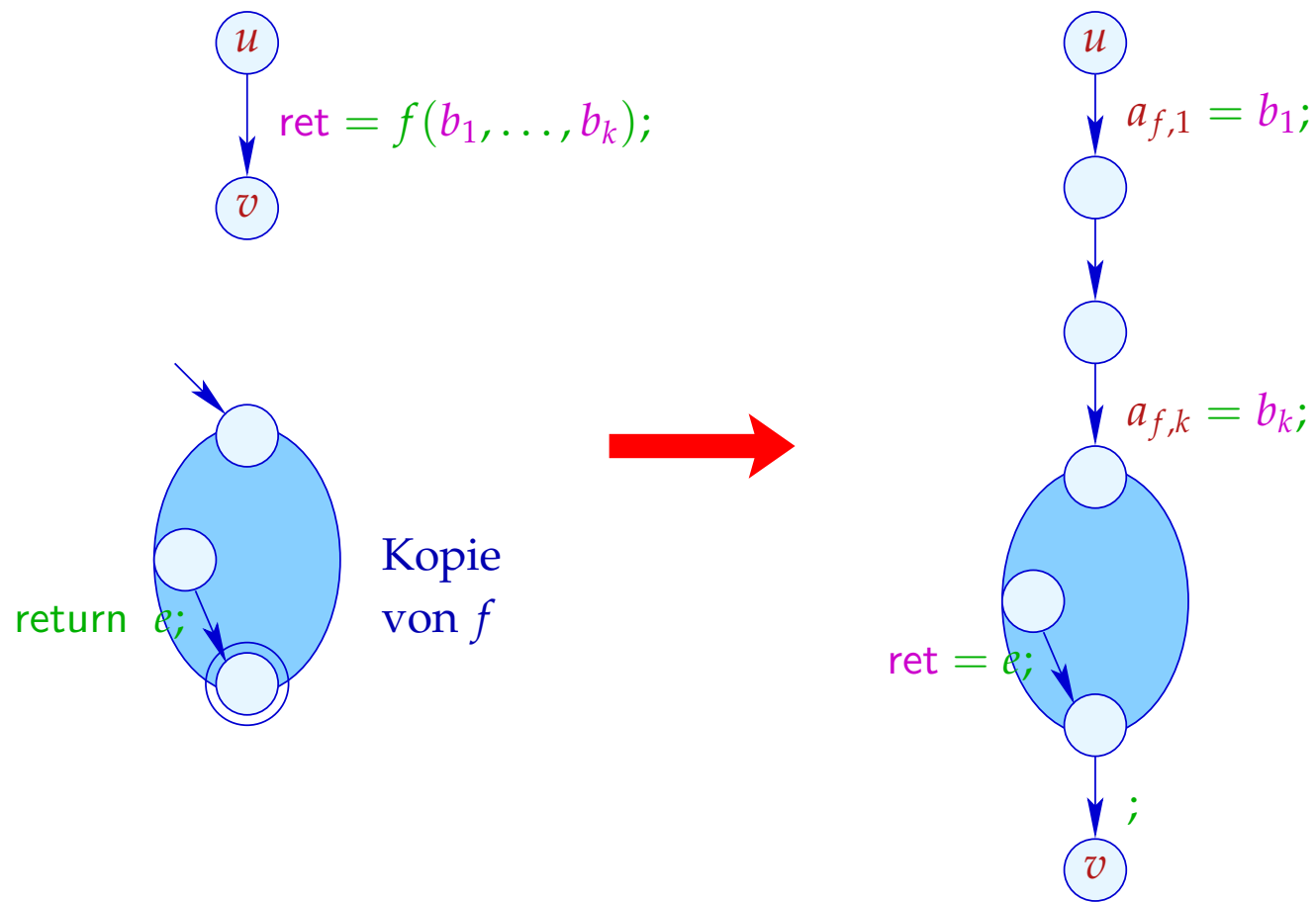
- Die Knoten sind die Funktionen.
- Eine Kante geht von  $g$  nach  $h$ , sofern der Rumpf von  $g$  einen Aufruf von  $h$  enthält.

## Strategien für Inlining:

- Kopiere nur **Blatt**-Funktionen ein, d.h. solche ohne weitere Aufrufe :-)
- Kopiere sämtliche nicht-rekursiven Funktionen ein!

... wir betrachten hier nur Blatt-Funktionen ;-)

# Transformation 10:



## Beachte:

- Blatt-Funktionen benutzen keine Variablen  $b_i$  :-)
- Die Variable  $ret$  der ein-kopierten Funktion wird mit der Variablen  $ret$  der aufrufenden identifiziert.  
Das spart uns eine Umspeicherung :-)
- Die **Nop**-Kante können wir ebenfalls einsparen, da der *stop*-Knoten von  $f$  selbst keine ausgehenden Kanten hat ...
- Die  $a_{f,i}$  sind die Kopien der aktuellen Parameter von  $f$ .
- Eigentlich müssten wir alle anderen Variablen von  $f$  noch mit 0 initialisieren :-)



## 2. Idee: Beseitigung von Endrekursion

Betrachte das folgende Beispiel:

```
f(a1, a2) {  
    if (a2 ≤ 1) return a1;  
    b1 = a1 · a2;  
    b2 = a1 - 1;  
    ret = f(b1, b2);  
    return ret;  
}
```

Der Funktionsaufruf im Rumpf liefert das Ergebnis.

Nach dem Funktionsaufruf gibt es im Rumpf nichts mehr zu tun.

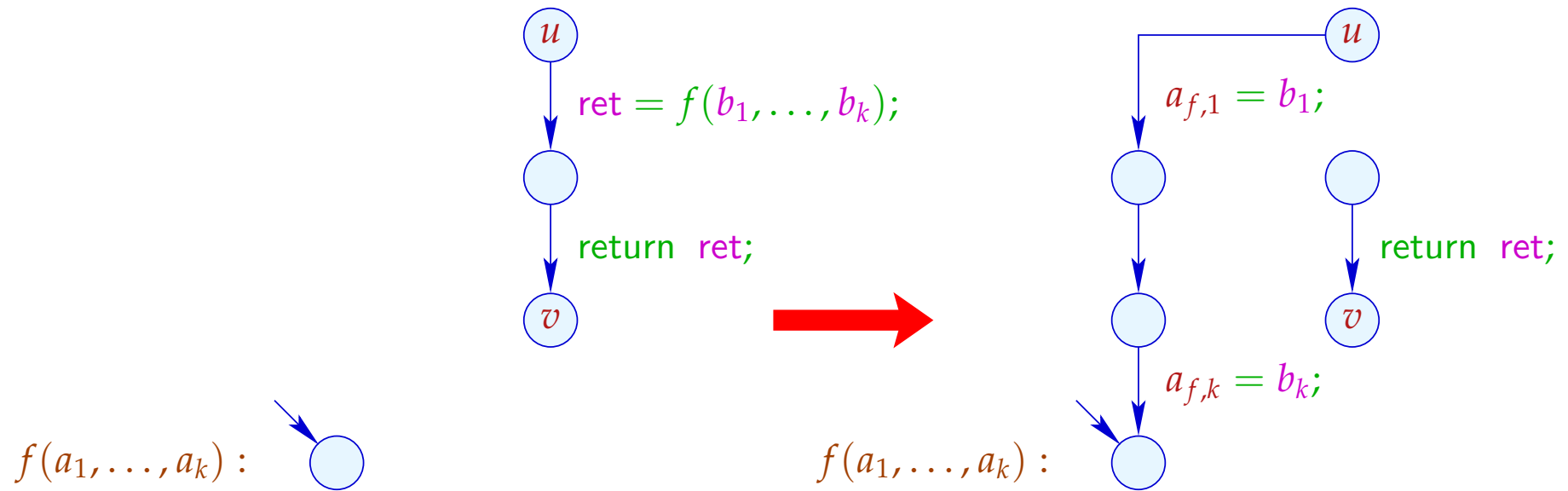
⇒ Wir könnten ihn **direkt anspringen** :-)

... das liefert im Beispiel:

```
f(a1, a2) {  
  _f : if (a2 ≤ 1) return a1;  
      b1 = a1 · a2;  
      b2 = a1 - 1;  
      a1 = b1;  
      a2 = b2;  
      goto _f;  
}
```

```
// Das funktioniert, weil wir Referenzen auf Variablen  
// verbieten.
```

## Transformation 11:



## Achtung:

- Diese Optimierung ist besonders wichtig bei Programmiersprachen ohne Iterationskonstrukte !!!
- Duplizieren von Code ist nicht erforderlich :-)
- Variablen brauchen nicht umbenannt zu werden :-)
- Die Optimierung hilft auch bei nicht-rekursiven Endaufrufen :-)
- Der entstehende Code kann Sprünge aus dem Rumpf einer Funktion in eine andere enthalten ???

## Exkurs 4: Interprozedurale Analyse

Bisher können wir nur jede Funktion einzeln analysieren.

- Die Kosten halten sich in Grenzen :-)
- Die Techniken funktionieren auch bei getrennter Übersetzung :-)
- An Funktionsaufrufen müssen wir das Schlimmste annehmen :-((
- Konstantenpropagation funktioniert nur für lokale Konstanten :-(((

Frage:

Wie analysiert man rekursive Programme ???

## Beispiel:

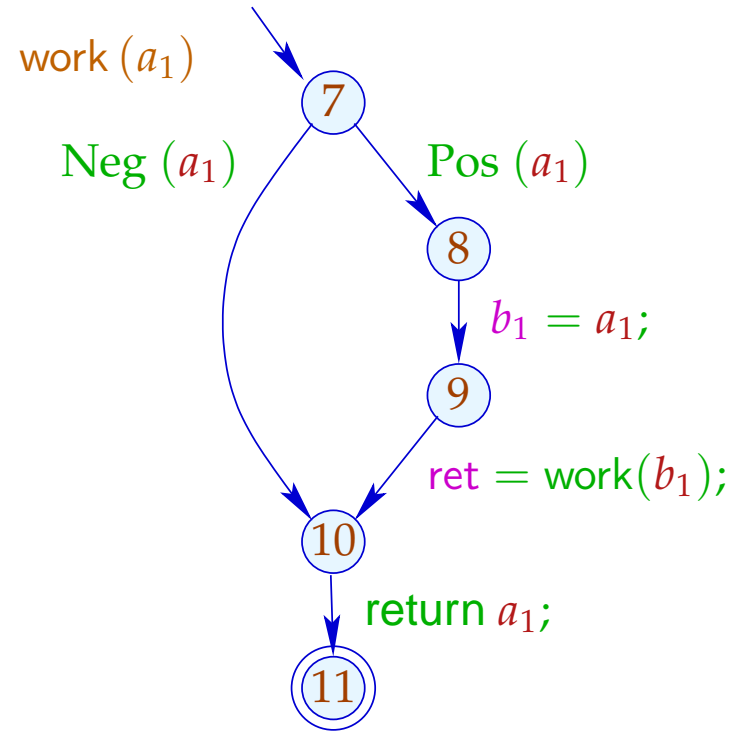
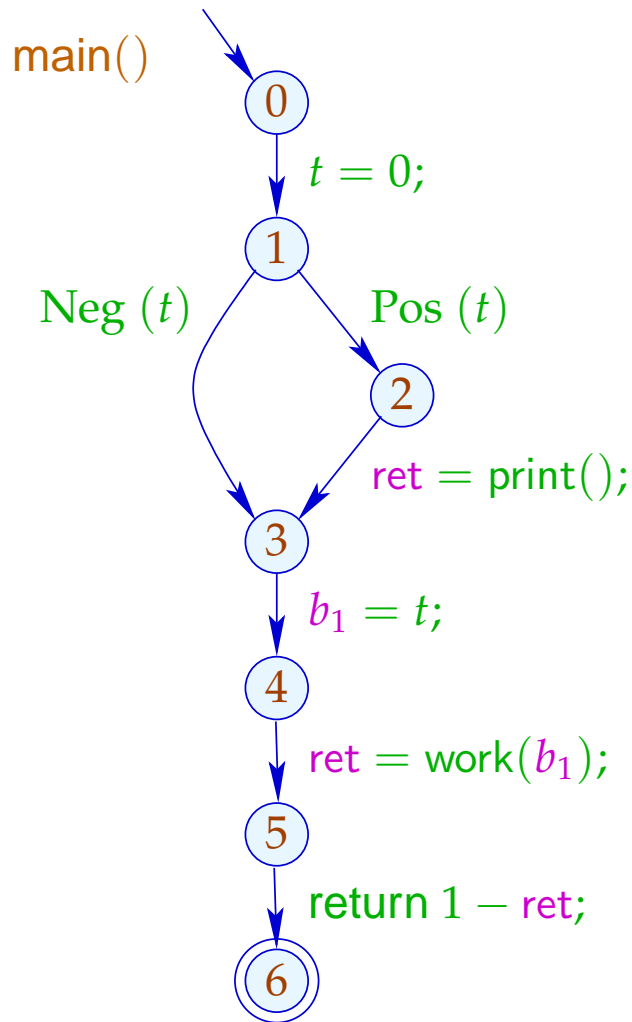
## Konstantenpropagation

```
main() {  
    t = 0;  
    if (t) print();  
    b1 = t;  
    ret = work(b1);  
    return 1 - ret;  
}
```

```
work(a1) {  
    if (a1) {  
        b1 = a1;  
        work(b1);  
    }  
    return a1;  
}
```

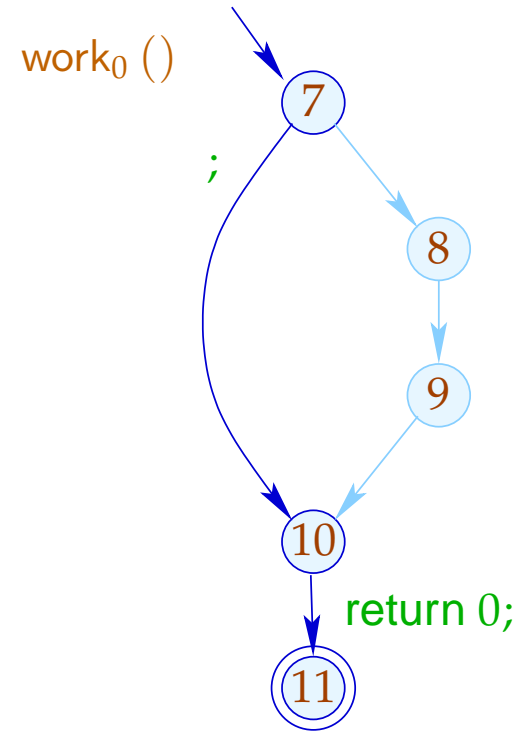
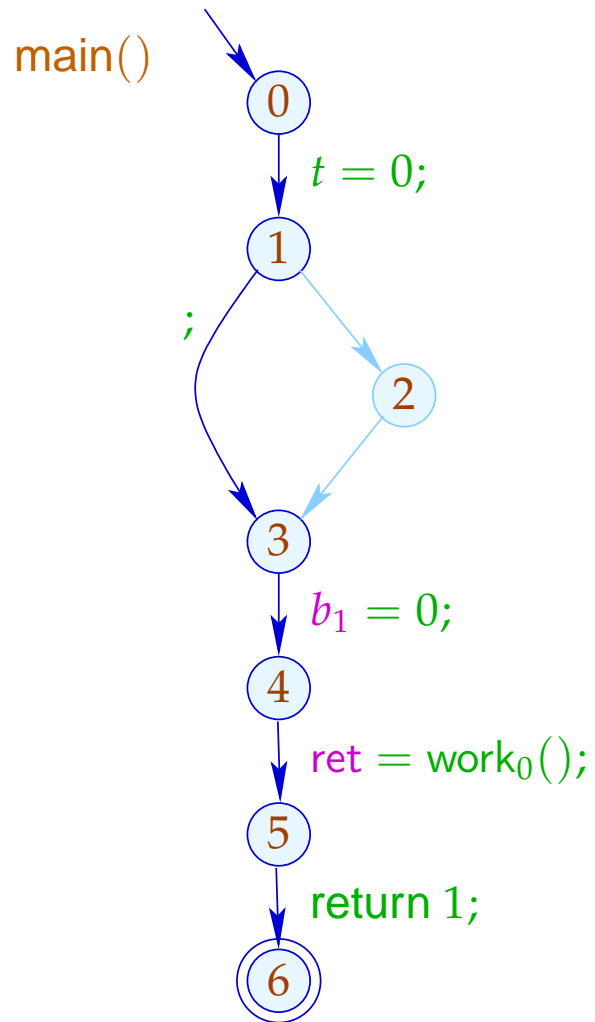
Beispiel:

# Konstantenpropagation



Beispiel:

# Konstantenpropagation





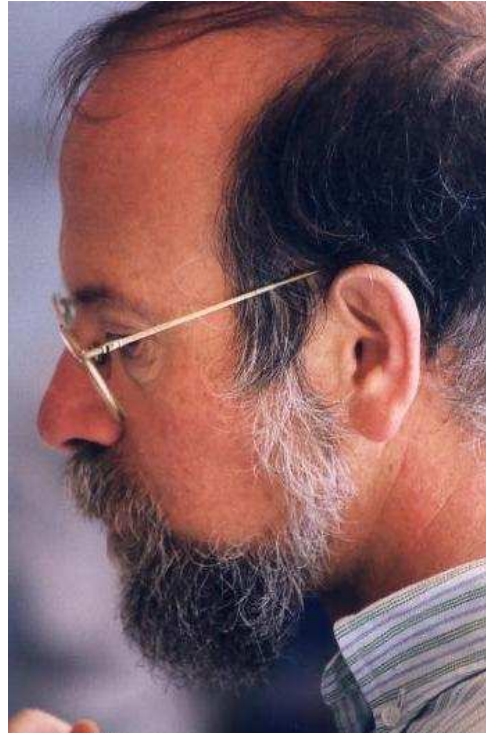
(1) **Funktionaler Ansatz:**

Sei  $\mathbb{D}$  ein vollständiger Verband von (abstrakten) Zuständen.

Idee:

Repräsentiere den Effekt von  $f()$  durch eine Funktion:

$$[[f]]^\# : \mathbb{D} \rightarrow \mathbb{D}$$



Micha Sharir, Tel Aviv University



Amir Pnueli, Weizmann Institute

Um den Effekt einer Aufrufskante  $k = (u, \text{ret} = f(b_1, \dots, b_k);, v)$  zu ermitteln, benötigen wir abstrakte Funktionen:

$$\begin{aligned} \text{enter}_f^\# & : \mathbb{D} \rightarrow \mathbb{D} \\ \text{combine}^\# & : \mathbb{D}^2 \rightarrow \mathbb{D} \end{aligned}$$

Damit erhalten wir:

$$\llbracket k \rrbracket^\# D = \text{combine} (D, \llbracket f \rrbracket^\# (\text{enter}_f^\# D))$$

... für Konstantenpropagation:

$$\mathbb{D} = (\text{Vars} \rightarrow \mathbb{Z}^\top)_\perp$$

$$\text{enter}_f^\# D = \begin{cases} \perp & \text{falls } D = \perp \\ \{a_i \mapsto (D b_i) \mid i = 1, \dots, k\} & \text{sonst} \end{cases}$$

$$\text{combine}^\# (D_1, D_2) = \begin{cases} \perp & \text{falls } D_1 = \perp \vee D_2 = \perp \\ D_1 \oplus \{\text{ret} \mapsto (D_2 \text{ret})\} & \text{sonst} \end{cases}$$

Um die Effekte  $\llbracket f \rrbracket^\#$  zu ermitteln, stellen wir ein Constraint-System über dem vollständigen Verband  $\mathbb{D} \rightarrow \mathbb{D}$  auf:

$$\begin{aligned}
 \llbracket v \rrbracket^\# &\sqsupseteq \text{Id} && v \text{ Eintrittspunkt} \\
 \llbracket v \rrbracket^\# &\sqsupseteq \llbracket k \rrbracket^\# \circ \llbracket u \rrbracket^\# && k = (u, \_, v) \text{ Kante} \\
 \llbracket f \rrbracket^\# &\sqsupseteq \llbracket \text{stop}_f \rrbracket^\# && \text{stop}_f \text{ Endpunkt von } f
 \end{aligned}$$

$\llbracket v \rrbracket^\# : \mathbb{D} \rightarrow \mathbb{D}$  beschreibt den Effekt aller Präfixe der Berechnungswälder  $w$  einer Funktion, die vom Eintrittspunkt nach  $v$  führen :-)

## Probleme:

- Wie beschreibt man eine Funktion  $f : \mathbb{D} \rightarrow \mathbb{D} ???$
- Ist  $\#\mathbb{D} = \infty$ , hat  $\mathbb{D} \rightarrow \mathbb{D}$  **unendliche** aufsteigende Ketten  
:-)

## Vereinfachung: Kopier-Konstanten

- Bedingungen interpretieren wir wie ein  $; :-)$
- Wir behandeln exakt nur Zuweisungen  $x = e;$  mit  
 $e \in \text{Vars} \cup \mathbb{Z} :-)$

## Beobachtung:

→ Die Effekte von Zuweisungen sind:

$$\llbracket x = e; \rrbracket^\# D = \begin{cases} D \oplus \{x \mapsto c\} & \text{falls } e = c \in \mathbb{Z} \\ D \oplus \{x \mapsto (D \ y)\} & \text{falls } e = y \in \text{Vars} \\ D \oplus \{x \mapsto \top\} & \text{sonst} \end{cases}$$

→ Sei  $\mathbb{V}$  die (endliche !!!) Menge der **konstanten** rechten Seiten. Dann haben Variablen stets Werte aus  $\mathbb{V}^\top$  :-))

→ Die auftretenden Effekte sind enthalten in

$$\mathbb{D}_f \rightarrow \mathbb{D}_f \quad \text{mit} \quad \mathbb{D}_f = (\text{Vars} \rightarrow \mathbb{V}^\top)_\perp$$

→ Dieser Verband ist riesig, aber **endlich !!!**