Diese operationelle Semantik ist einigermaßen realistisch :-)

Kosten eines Prozedur-Aufrufs:

Vor Betreten des Rumpfs:

- Anlegen eines Kellerrahmens;
- Retten der Register;
- Retten der Fortsetzungsaddresse;
- Anspringen des Rumpfs.

Bei Beenden des Aufrufs:

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

$\Longrightarrow$ ... ziemlich teuer !!!

1. Idee: Inlining

Kopiere den Funktionsrumpf an jede Aufrufstelle !!!

Beispiel:

$$
\begin{array}{l}
abs\ (a_1)\ \{ \\
\qquad b_1 = a_1; \\
\qquad b_2 = -a_1; \\
\qquad \mathsf{ret} = max\ (b_1, b_2); \\
\qquad \mathsf{return}\ \ \mathsf{ret}; \\
\}
\end{array}
\qquad
\begin{array}{l}
max\ (a_1, a_2)\ \{ \\
\qquad \mathsf{if}\ \ (a_1 < a_2)\ \ \mathsf{return}\ \ a_2; \\
\qquad \mathsf{return}\ \ a_1; \\
\}
\end{array}
$$

... liefert:

$$
\begin{array}{ll}
abs\ (a_1)\ \{ & \\
 & b_1 = a_1; \\
 & b_2 = -a_1; \\
 & a_1 = b_1; \\
 & a_2 = b_2; \\
 & \boxed{\begin{array}{l} \text{if}\ \ (a_1 < a_2)\ \ \{\ \ \text{ret} = a_2;\ \ \text{goto}\ \ _max;\ \ \} \\ \text{ret} = a_1;\ \ \text{goto}\ \ _max; \end{array}} \\
_max: & \text{return}\ \ \text{ret}; \\
\} &
\end{array}
$$

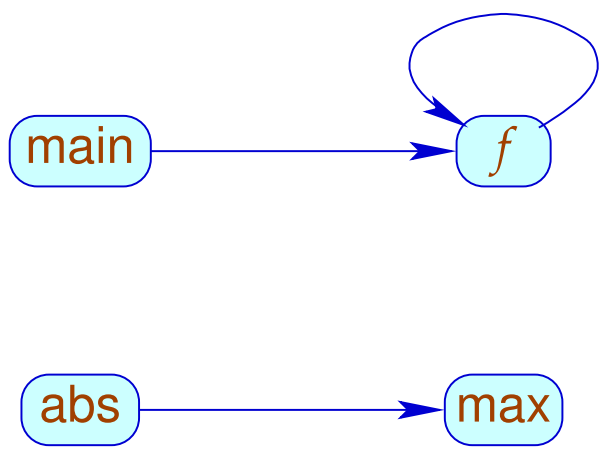
Probleme:

- 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:

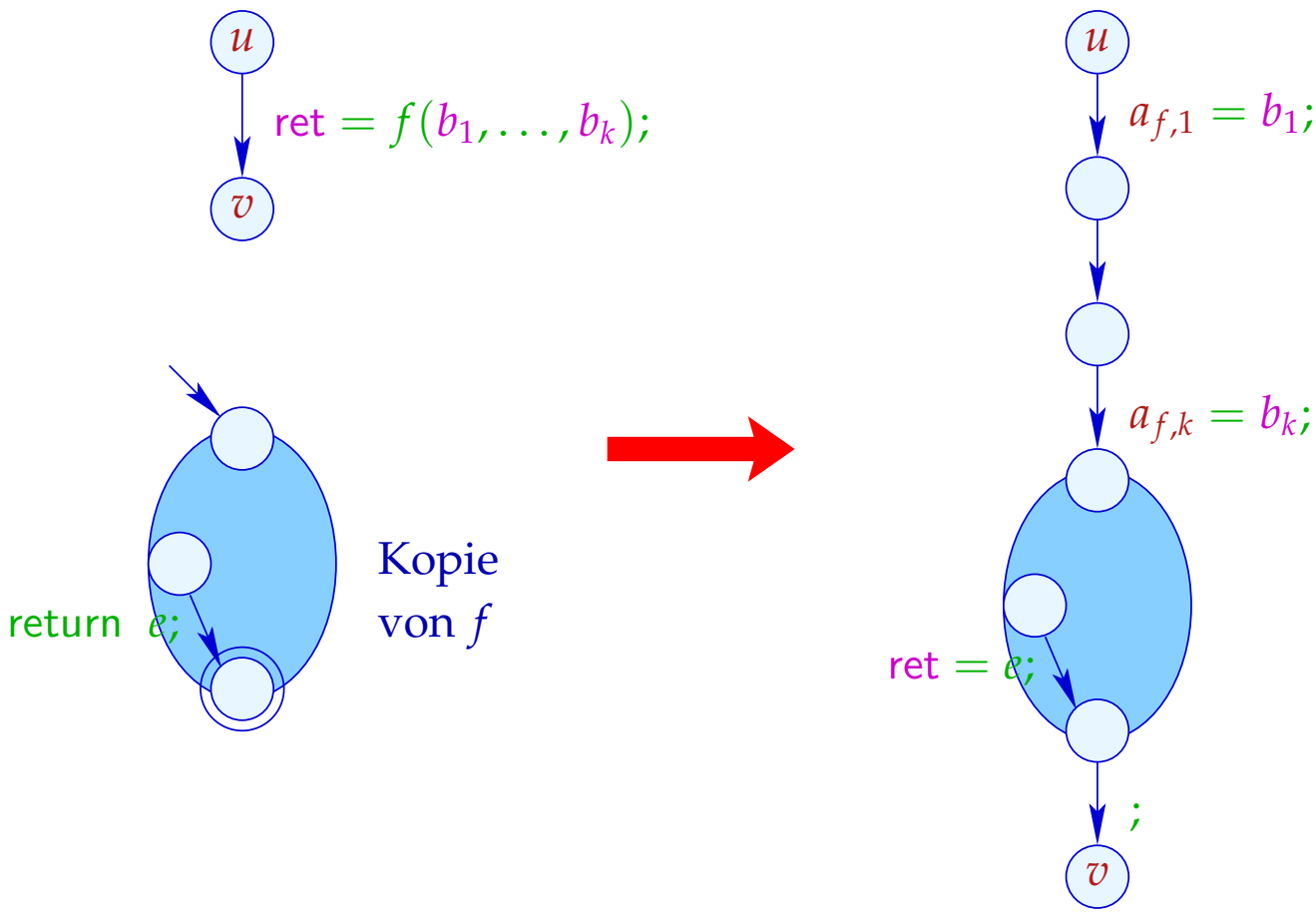
- Die Knoten sind die Funktionen.
- Eine Kante geht von g nach h, sofern der Rumpf von g einen Aufruf von h enthält.

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:

$$
\begin{array}{l}
f\,(a_1, a_2)\ \ \{ \\
\qquad \text{if}\ \ (a_2 \leq 1)\ \ \text{return}\ \ a_1; \\
\qquad b_1 = a_1 \cdot a_2; \\
\qquad b_2 = a_1 - 1; \\
\qquad \text{ret} = f\,(b_1, b_2); \\
\qquad \text{return}\ \ \text{ret}; \\
\}
\end{array}
$$

Der Funktionsaufruf im Rumpf liefert das Ergebnis.

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

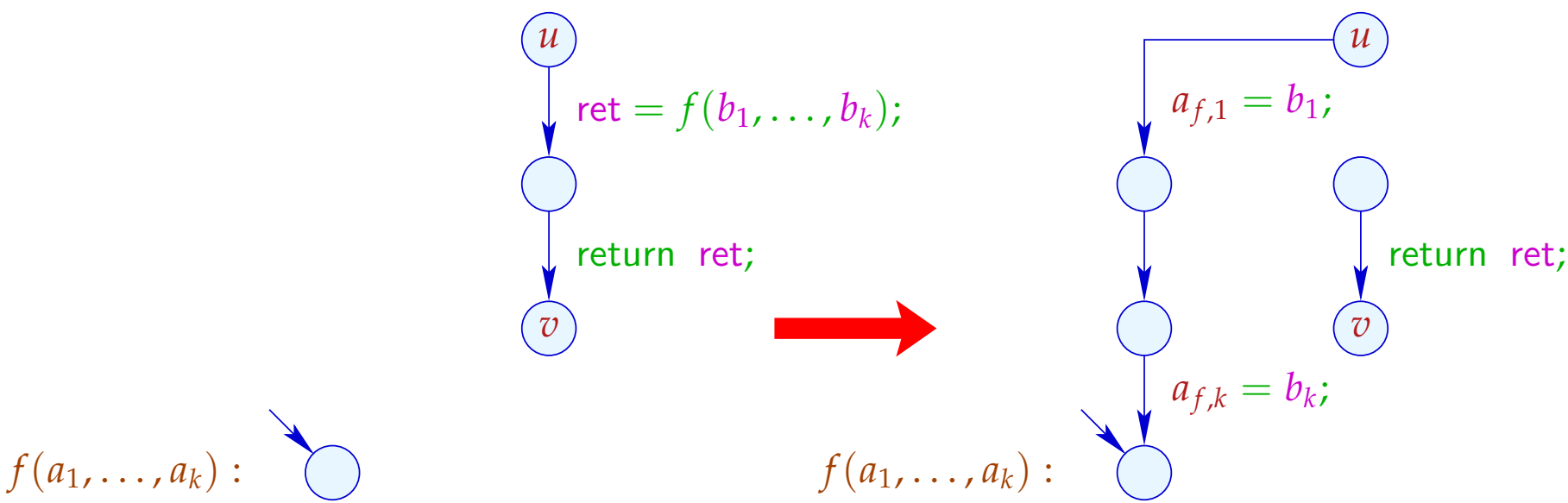
$\Longrightarrow$ Wir könnten ihn direkt anspringen :-)

... das liefert im Beispiel:

$$
\begin{array}{l}
f\,(a_1, a_2)\ \ \{ \\
\quad _f:\ \ \text{if}\ \ (a_2 \leq 1)\ \ \text{return}\ \ a_1; \\
\qquad\qquad b_1 = a_1 \cdot a_2; \\
\qquad\qquad b_2 = a_1 - 1; \\
\qquad\qquad a_1 = b_1; \\
\qquad\qquad a_2 = b_2; \\
\qquad\qquad \text{goto}\ \ _f; \\
\}
\end{array}
$$

// 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:

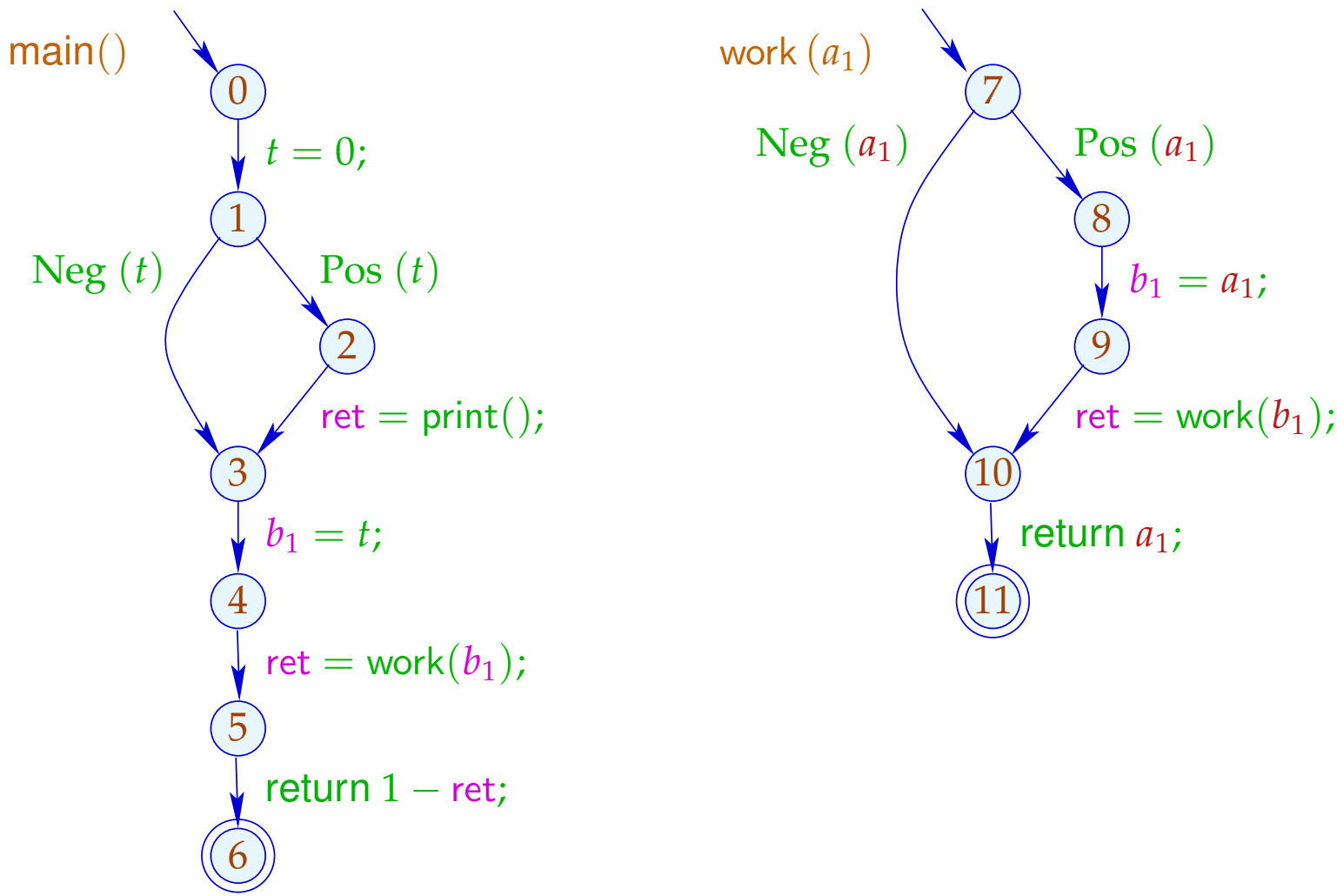
Wie analysiert man rekursive Programme ???

Beispiel: Konstantenpropagation

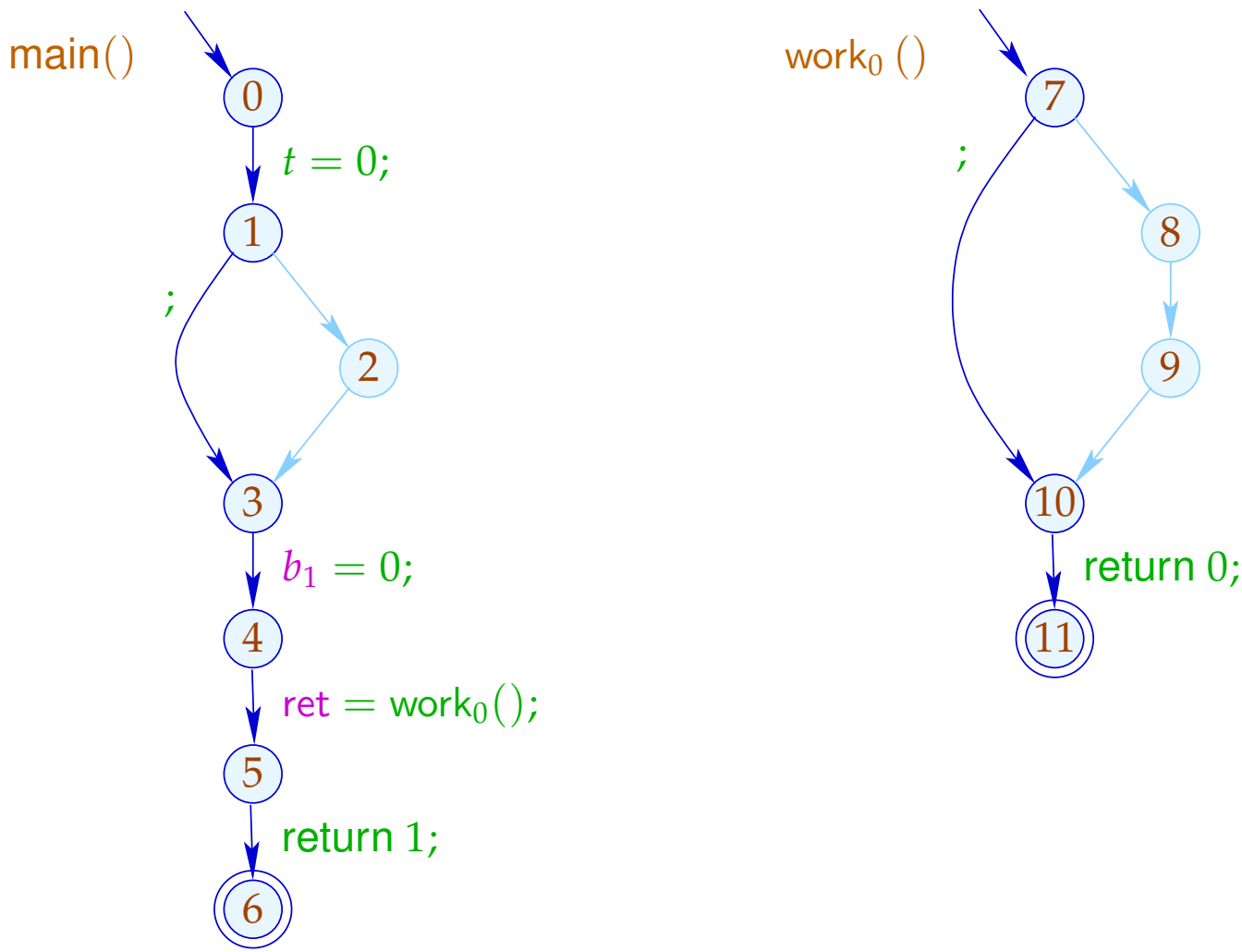
```
main() {
    t = 0;
    if (t) print();
    b1 = t;
    ret = work(b1);
    return 1 − ret;
}
```

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

Beispiel: Konstantenpropagation



Beispiel: Konstantenpropagation



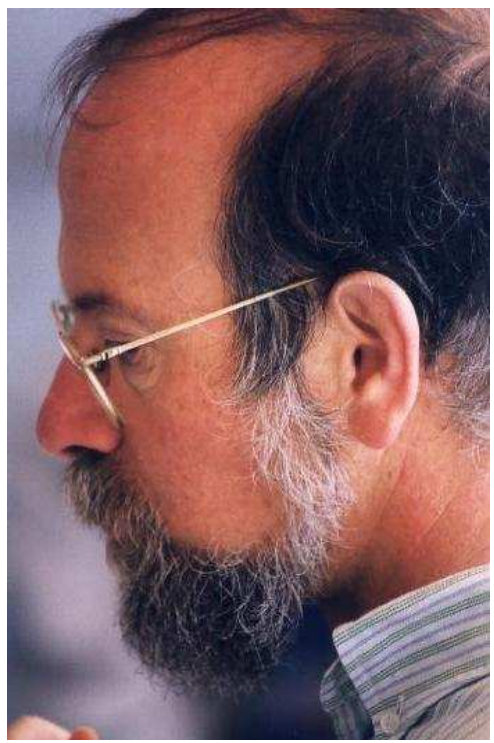
(1) Funktionaler Ansatz:

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

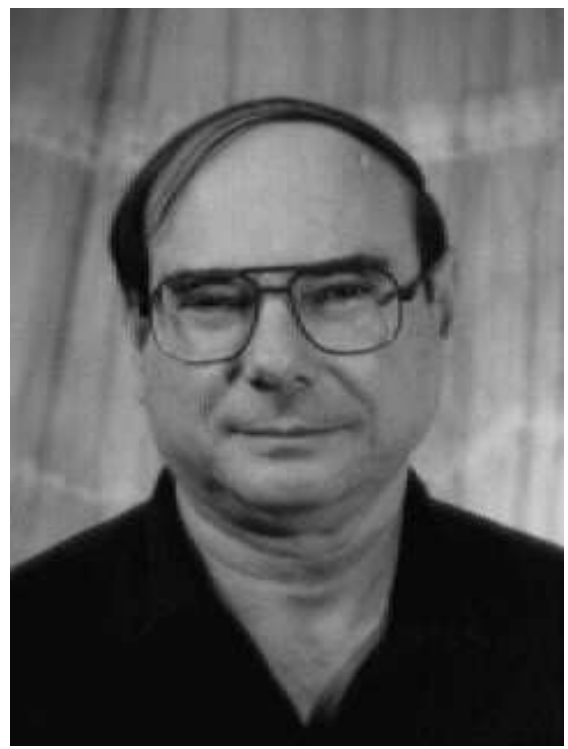
Idee:

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

$$[\![f]\!]^\sharp \; : \; \mathbb{D} \to \mathbb{D}$$



Micha Sharir, Tel Aviv University



Amir Pnueli, Weizmann Institute

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

$$\begin{array}{lcl} \mathsf{enter}^\sharp_f & : & \mathbb{D} \to \mathbb{D} \\ \mathsf{combine}^\sharp & : & \mathbb{D}^2 \to \mathbb{D} \end{array}$$

Damit erhalten wir:

$$[\![k]\!]^\sharp\ D \quad = \quad \mathsf{combine}^\sharp\ (D, [\![f]\!]^\sharp\ (\mathsf{enter}^\sharp_f\ D))$$

... für Konstantenpropagation:

$$\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^{\top})_{\bot}$$

$$\mathsf{enter}^{\sharp}_{f}\ D = \begin{cases} \bot & \text{falls} \quad D = \bot \\ \{a_i \mapsto (D\ b_i) \mid i = 1, \ldots, k\} & \text{sonst} \end{cases}$$

$$\mathsf{combine}^{\sharp}\ (D_1, D_2) = \begin{cases} \bot & \text{falls} \quad D_1 = \bot \vee D_2 = \bot \\ D_1 \oplus \{\mathsf{ret} \mapsto (D_2\ \mathsf{ret})\} & \text{sonst} \end{cases}$$

Um die Effekte $[\![f]\!]^\sharp$ zu ermitteln, stellen wir ein Constraint-System über dem vollständigen Verband $\mathbb{D} \to \mathbb{D}$ auf:

$$
\begin{array}{lll}
[\![v]\!]^\sharp & \sqsupseteq & \mathsf{Id} \qquad\qquad v \quad \text{Eintrittspunkt} \\
[\![v]\!]^\sharp & \sqsupseteq & [\![k]\!]^\sharp \circ [\![u]\!]^\sharp \qquad k = (u, _, v) \quad \text{Kante} \\
[\![f]\!]^\sharp & \sqsupseteq & [\![stop_f]\!]^\sharp \qquad stop_f \quad \text{Endpunkt von} \quad f
\end{array}
$$

$[\![v]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ beschreibt den Effekt aller Präfixe der Berechnungswälder w einer Funktion, die vom Eintrittspunkt nach v führen :-)

Probleme:

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

Vereinfachung: Kopier-Konstanten

$\to$ Bedingungen interpretieren wir wie ein $;$:-)

$\to$ Wir behandeln exakt nur Zuweisungen $x = e;$ mit $e \in Vars \cup \mathbb{Z}$:-)

Beobachtung:

→ Die Effekte von Zuweisungen sind:

$$[\![x = e;]\!]^\sharp\, D \quad = \quad \begin{cases} D \oplus \{x \mapsto c\} & \text{falls} \quad e = c \in \mathbb{Z} \\ D \oplus \{x \mapsto (D\ y)\} & \text{falls} \quad e = y \in \mathit{Vars} \\ D \oplus \{x \mapsto \top\} & \text{sonst} \end{cases}$$

→ 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

$$\mathbb{D}_f \to \mathbb{D}_f \qquad \text{mit} \qquad \mathbb{D}_f = (\mathit{Vars} \to \mathbb{V}^\top)_\perp$$

→ Dieser Verband ist riesig, aber endlich !!!

Verbesserung:

$\rightarrow$ Nicht alle Funktionen aus $\mathbb{D}_f \rightarrow \mathbb{D}_f$ kommen wirklich vor :-)

$\rightarrow$ Alle vorkommenden Funktionen $\lambda D. \bot \neq M$ sind von der Form:

$$\begin{array}{lcll} M & = & \{x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} y) \mid x \in \mathit{Vars}\} & \text{wobei:} \\ M\ D & = & \{x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} D\ y) \mid x \in \mathit{Vars}\} & \text{für} \quad D \neq \bot \end{array}$$

$\rightarrow$ Sei $\mathbb{M}$ die Menge aller dieser Funktionen. Dann gilt für $M_1, M_2 \in \mathbb{M}$ $(M_1 \neq \lambda D. \bot \neq M_2)$:

$$(M_1 \sqcup M_2)\ x \quad = \quad (M_1\ x) \sqcup (M_2\ x)$$

$\rightarrow$ Für $k = \#\mathit{Vars}$ hat $\mathbb{M}$ die Höhe $(k+1) \cdot k + 1$:-)