Was bisher geschah

Turing = Menge aller durch eine DTM berechenbaren Funktionen

goto-Programme:

- Syntax:
  - Befehle B: `Inc i, Dec i, Goto k, GotoZ i k, Stop`
  - goto-Programm $p \in B^*$ (endliche Folge von Befehlen)
- Semantik:
  - Konfigurationen $C \subset \mathbb{N} \times S$ (Befehlszähler, Speicherbelegung)
  - Übergangsrelation $\text{step}_p \subseteq C \times C$
  - Start-, Finalkonfigurationen

  goto-Programm p berechnet partielle Funktion $f_p : \mathbb{N}^n \to \mathbb{N}$
- goto-Programme für Basisfunktionen: Konstanten, Projektionen, Nachfolger, Vorgänger, Test $s(i) = 0$
- Verknüpfung von goto-Programmen durch Operationen: Verkettung, Verzweigung

GOTO = Menge aller durch goto-Programm berechenbaren Funktionen

Satz: GOTO = Turing

Strukturierte Programme – Motivation

Goto-Programme sind flach (Listen von Befehlen),
haben keine sichtbare Struktur.
Das ist gut für die Hardware, schlecht für den Programmierer.

Struktur = Hierarchie = Baum

Programme sind ab jetzt Bäume. (entspricht etwa
dem Schritt von Assembler/Fortran zu Algol, $\approx$ 1960)

Bemerkung:
Das ist immer noch imperative Programmierung,
also immer noch unübersichtlich für den Programmierer
(Semantikdefinition verwendet Maschinenzustand,
dieser ist im Programm nicht sichtbar).

Ausweg: funktionale Programmierung (kein Zustand).

Syntax

Menge aller While-Programme P (induktive Definition):

IA: elementare Programme $\mathsf{Inc}\, i$, $\mathsf{Dec}\, i$ mit $i \in \mathbb{N}$,
leeres Programm: Skip

IS: zusammengesetzte Programme
Sind $p, q \in P$ und $i \in \mathbb{N}$, dann auch

- Nacheinanderausführung: $\mathsf{Seq}(p, q) \in P$
- Verzweigung: $\mathsf{IfZ}(i, p, q) \in P$
- Wiederholung: $\mathsf{While}(i, p) \in P$

(kein Stop, kein Goto)

Beispiel:

- $\mathsf{While}(1, \mathsf{Seq}(\mathsf{Dec}(1), \mathsf{Inc}(0)))$
- autotool-Syntax: `While 1 (Seq (Dec 1) (Inc 0))`

Semantik (Prinzip)

Semantik eines Programms $p \in P$
ist eine Relation (genauer: partielle Funktion)
$\text{sem}_p \subseteq S \times S$ auf Speicherbelegungen.

big step semantics (ein Schritt)

Achtung:
Es gibt keinen Befehlszähler, diese Rolle übernimmt der Index p.

Semantik von While-Programmen

elementare Programme:

$$
\begin{aligned}
\mathrm{sem}_{\mathsf{Skip}} &= I_{(\mathbb{N}^{\mathbb{N}})} = \{(s,s) \mid s : \mathbb{N} \to \mathbb{N}\} \\
\mathrm{sem}_{\mathsf{Inc}(i)} &= \{(s, s[i := s(i)+1]) \mid s : \mathbb{N} \to \mathbb{N}\} \\
\mathrm{sem}_{\mathsf{Dec}(i)} &= \{(s, s[i := \max(0, s(i)-1]) \mid s : \mathbb{N} \to \mathbb{N}\}
\end{aligned}
$$

zusammengesetzte Programme p:

$$
\begin{aligned}
\mathrm{sem}_{\mathsf{Seq}(p_1,p_2)} &= \mathrm{sem}_{p_1} \circ \mathrm{sem}_{p_2} \\
&= \left\{(s_1,s_2) \mid \exists s' : \left(s_1,s'\right) \in \mathrm{sem}_{p_1} \wedge \left(s',s_2\right) \in \mathrm{sem}_{p_2}\right\} \\
\mathrm{sem}_{\mathsf{IfZ}(i,p_1,p_2)} &= \left\{(s_1,s_2) \mid \begin{array}{l} (s_1(i)=0 \wedge (s_1,s_2) \in \mathrm{sem}_{p_1}) \\ \vee (s_1(i)>0 \wedge (s_1,s_2) \in \mathrm{sem}_{p_2}) \end{array}\right\} \\
\mathrm{sem}_{\mathsf{While}(i,q)} &= \left\{(s_1,s_2) \mid \begin{array}{l} (s_1(i)=0 \wedge (s_1=s_2)) \\ \vee (s_1(i)>0 \wedge (s_1,s_2) \in \mathrm{sem}_{\mathsf{Seq}(q,p)}(s_1,s_2)) \end{array}\right\}
\end{aligned}
$$

Äquivalenz von Programmen

Programme p und q sind äquivalent ($p \equiv q$) gdw. $\mathrm{sem}_p = \mathrm{sem}_q$.

Beispiele:

- $\mathrm{Seq}(\mathrm{Skip}, p) \equiv p \equiv \mathrm{Seq}(p, \mathrm{Skip})$
  für erste Äquivalenz z.z.: $\mathrm{sem}_{\mathrm{Seq}(\mathrm{Skip},p)} = \mathrm{sem}_p$

$$\begin{aligned} \mathrm{sem}_{\mathrm{Seq}(\mathrm{Skip},p)} &\overset{(\mathrm{Seq})}{=} \mathrm{sem}_{\mathrm{Skip}} \circ \mathrm{sem}_p \\ &\overset{(\mathrm{Skip})}{=} I_{\mathbb{N}^{\mathbb{N}}} \circ \mathrm{sem}_p \\ &\overset{I_{\mathbb{N}^{\mathbb{N}}}\ \text{bzgl.}\ \circ\ \text{neutral}}{=} \mathrm{sem}_p \end{aligned}$$

- $\mathrm{Seq}(\mathrm{Seq}(p, q), r) \equiv \mathrm{Seq}(p, \mathrm{Seq}(q, r))$
- $\mathrm{While}(i, p) \equiv \mathrm{IfZ}(i, \mathrm{Skip}, \mathrm{Seq}(p, \mathrm{While}(i, p)))$

ÜA: IfZ wird nicht benötigt, lässt sich simulieren

While-berechenbare Funktionen

- initiale Speicherbelegung $I(x)$ für Eingabe $x \in \mathbb{N}^n$:
  $s(i) =$ wenn $1 \leqslant i \leqslant n$, dann x_i, sonst 0.
- Programm p berechnet partielle Funktion $f : \mathbb{N}^n \to \mathbb{N}$:
  $\forall x \in \mathbb{N}^n : f(x) = y$ gdw. $\exists s : \mathrm{sem}_p(I(x), s) \wedge y = s(0)$.
- jede so berechenbare partielle Fkt. heißt While-berechenbar.

WHILE = Menge aller While-berechenbaren Funktionen

- Die üblichen elementaren Funktionen sind $\in$ WHILE.
  (Autotool)
- WHILE ist abgeschlossen unter Substitution, z.B.
  $(f \in \mathrm{WHILE} \wedge g \in \mathrm{WHILE}) \to (x \mapsto f(g(x)) \in \mathrm{WHILE}$ (ÜA)

Beziehungen zwischen Goto- und While-Berechenbarkeit

Satz: WHILE = GOTO

äquivalente Aussage:
Für jede partielle Funktion f gilt:
f ist While-berechenbar gdw. f ist Goto-berechenbar.

praktisches Argument für „$\subseteq$“:
ist genau die Aufgaben von Compilern
(etwa von C nach Assembler/Maschinensprache)

(siehe Modul „Compilerbau“)

WHILE $\subseteq$ GOTO (Prinzip)

Übersetzer:
compile : $\mathbb{N} \times \text{WHILE} \to \mathbb{N} \times \text{GOTO}$
wobei compile$(a, p) = (e, q)$ bedeutet:

- das While-Programm p wird übersetzt
- in ein äquivalentes Goto-Programm q,
- das auf Adresse a beginnt
- und auf $e - 1$ endet (d.h. $e = a + |q|$)

dabei bedeutet „Äquivalenz“:
$\forall p \forall a \in \mathbb{N}$ mit compile$(a, p) = (e, q)$:
$\forall s_1, s_2 : (s_1, s_2) \in \text{sem}_p \leftrightarrow ((a, s_1), (e, s_2)) \in \text{step}_q^*$

WHILE $\subseteq$ GOTO (elementar, Seq)

einfache Programme:

- $\text{compile}(a, \text{Skip}) = (a, [])$
- $\text{compile}(a, \text{Inc}(i)) = (a + 1, [\text{Inc}(i)])$.
- $\text{compile}(a, \text{Dec}(i)) = (a + 1, [\text{Dec}(i)])$.

zusammengesetzte Programme:

- $\text{compile}(a, \text{Seq}(p, q)) = (e, p' \circ q')$, wobei $\text{compile}(a, p) = (m, p')$ und $\text{compile}(m, q) = (e, q')$

WHILE $\subseteq$ GOTO (While, IfZ)

IfZ:

```
compile (_, IfZ i p1 p2) =>
  A: GotoZ i M
     compile (_, p2) ;
     Goto E ;
  M: compile (_, p1);
  E: ...
```

While:

```
compile (_, While i p) =>
  A: GotoZ i E
     compile (A+1, p) ;
     Goto A ;
  E: ...
```

WHILE $\subseteq$ GOTO – Korrektheit Inc i

Diese Definition von compile(a, p) erfüllt die Spezifikation:
$\forall p \in \text{While}, a \in \mathbb{N}$ gilt für $(e, q) = \text{compile}(a, p)$:
$\forall s, s' : (s, s') \in \text{sem}_p$ gdw. $((a, s), (e, s')) \in \text{step}_q^*$

Beweis der Korrektheit für $p = \text{Inc}\, i$:
$\text{compile}(a, p) = (e, q)$ mit $e = a + 1$ und $q = [\text{Inc}\, i]$

$$
\begin{aligned}
\forall s, s' : \quad & (s, s') \in \text{sem}_p \\
\text{gdw.} \quad & (s, s') \in \text{sem}_{\text{Inc}\, i} \\
\text{gdw.} \quad & s' = s[i := s(i) + 1] \\
\text{gdw.} \quad & ((a, s), (\underbrace{a + 1}_{e}, \underbrace{s[i := s(i) + 1]}_{s'})) \in \text{step}^*_{[\text{Inc}\, i]} \\
\text{gdw.} \quad & ((a, s), (e, s')) \in \text{step}_q^*
\end{aligned}
$$

WHILE $\subseteq$ GOTO – Korrektheit While i p

Beweis der Korrektheit für $p = \text{While } i\ p'$:
$\text{compile}(a, p) = (e, q)$ mit $q = [\text{GotoZ } i\ e] \circ q' \circ [\text{Goto } a]$,
wobei $\text{compile}(a+1, p') = (e', q')$

- $\forall s, s'$ mit $s(i) = 0$:

$$
\begin{aligned}
& (s, s') \in \text{sem}_p \text{ gdw. } (s, s') \in \text{sem}_{\text{While } i\ p'} \text{ gdw. } s' = s \\
\text{gdw.} \quad & ((a, s), (e, s)) \in \text{step}_{[\text{GotoZ } i\ e]} \\
\text{gdw.} \quad & ((a, s), (e, s)) \in \text{step}^*_{[\text{GotoZ } i\ e] \circ q' \circ [\text{Goto } a]} \\
\text{gdw.} \quad & ((a, s), (e, s)) \in \text{step}^*_q
\end{aligned}
$$

- $\forall s, s'$ mit $s(i) > 0$:

$$
\begin{aligned}
& (s, s') \in \text{sem}_p \text{ gdw. } (s, s') \in \text{sem}_{\text{While } i\ p'} \\
\text{gdw.} \quad & (s, s') \in \text{sem}_{\text{Seq}(p', p)} \text{ gdw. } (s, s') \in \text{sem}_{p'} \circ \text{sem}_p \\
\text{gdw.} \quad & \exists s'' : (s, s'') \in \text{sem}_{p'} \wedge (s'', s') \in \text{sem}_p \\
\text{gdw.} \quad & \exists (e', s'') : \begin{pmatrix} & ((a+1, s), (e', s'')) \in \text{step}^*_{q'} \\ \wedge & ((e'+1, s''), (e, s')) \in \text{step}^*_q \end{pmatrix} \\
\text{gdw.} \quad & ((a, s), (e, s')) \in \text{step}^*_{[\text{GotoZ } i\ e] \circ q' \circ [\text{Goto } a]} \\
\text{gdw.} \quad & ((a, s), (e, s')) \in \text{step}^*_q
\end{aligned}
$$

WHILE $\subseteq$ GOTO (insgesamt)

Satz:
Für jedes While-Programm p existiert ein Goto-Programm q, welches dieselbe partielle Funktion berechnet wie p.

Beweis(-plan):

- $q = \text{compile}(0, p) \circ [\text{Stop}]$
- Aussage folgt aus Korrektheit bzgl. der Spezifikation
  $\forall p \in \text{WHILE}, a \in \mathbb{N}$ gilt für $(e, q) = \text{compile}(a, p)$:

$$\forall s_1, s_2 : (s_1, s_2) \in \text{sem}_p \quad \leftrightarrow \quad ((a, s_1), (e, s_2)) \in \text{step}_q^* \qquad (\ddot{\text{U}}\text{A})$$

GOTO $\subseteq$ WHILE

das scheint schwieriger:

- goto-Programm = Spaghetti-Code,
- while-Programm = strukturierter Code.

Es ist aber möglich und das erzeugte While-Programm hat eine besondere (einfache) Struktur, die später noch ausgenutzt wird (Kleene-Normalform-Theorem)

GOTO $\subseteq$ WHILE (Ansatz)

Eingabe: Goto-Programm p,
Ausgabe: äquivalentes While-program q

PC $c = 1 + \max\{i \mid \mathsf{Inc}\, i \in p \vee \mathsf{Dec}\, i \in p \vee \mathsf{GotoZ}\, i\ j \in p\}$
(erstes in p nicht benutzte Register)

Das nächste Register $c + 1 = h$ verwenden wir zum Anhalten.

Struktur von q ist:

```
Inc h;
While (h) {
  if (c == 0) { ... } else {
    if (c == 1) { ... } else {
      if (c == 2) { ... } else {
        ..                    else Skip
}
```

GOTO $\subseteq$ WHILE (Einzelheiten)

für Befehl p_i erzeuge: `if (c == i) q_i else ...` mit $q_i =$

- wenn $p_i \in \{\mathsf{Inc}\, r, \mathsf{Dec}\, r\}$, dann $\mathsf{Seq}(p_i, \mathsf{Inc}\, c)$
- wenn $p_i = \mathsf{Stop}$, dann $\mathsf{Dec}\, h$
- wenn $p_i = \mathsf{Goto}\, k$, dann $c := k$,
- wenn $p_i = \mathsf{GotoZ}\, r\ k$, dann `IfZ r (c := k) (Inc c)`

p erreicht Stop gdw. q hält. (ÜA)
Man beachte dabei auch den Fall $\mathsf{Goto}\, k$ mit $k \geqslant |p|$.

Hier wird `if (c == i)` und `c := 1` benutzt,
Implementierung durch While-Programme (ÜA)
(hier auch ohne Schleife möglich, warum?)

Normalform-Theorem für While

Vorige Konstruktion zeigt den
Satz:

- Zu jedem Goto-Programm existiert ein äquivalentes While-Programm (GOTO $\subseteq$ WHILE)
- mit genau einem While.

zusammen mit WHILE $\subseteq$ GOTO folgt

Satz: WHILE = GOTO

Satz (Kleene-Normalform für While-Programme):
Zu jedem While-Programm gibt es ein äquivalentes While-Programm mit genau einem While.

„äquivalent“= berechnet dieselbe partielle Funktion.