

Berechenbarkeit Vorlesung SS 15 (UL), WS 17

Johannes Waldmann

25. Januar 2018

1 Einleitung, Überblick

Inhalt und Ziel der Vorlesung

- grundlegende Begriffe, Prinzipien und Methoden aus der Algorithmentheorie und der Komplexitätstheorie
- ... zu einem tieferen Verständnis praktischer Problemstellungen.

(Quelle: Modulbeschreibung)

- was sind Algorithmen?
- wie hängen verschiedene Alg.-Definitionen zusammen?
- welche Probleme sind algorithmisch lösbar?
- ... mit welchem Ressourcenverbrauch?

Ist jede Funktion $\mathbb{N} \rightarrow \mathbb{N}$ berechenbar?

Nein! (Das ist ein wichtiges Resultat und wir sehen eine wichtige Beweismethode.)

- wir zählen alle Programmtexte auf (der Größe nach und innerhalb einer Größe lexikografisch), die totale Funktionen $\mathbb{N} \rightarrow \mathbb{N}$ realisieren.
- erhalten damit eine unendl. Folge von Funkt. f_0, f_1, \dots ,
jede berechnbare Fkt. kommt in dieser Folge vor (evtl. auch mehrfach, das ist egal)
- definiere $g : \mathbb{N} \rightarrow \mathbb{N} : x \mapsto f_x(x) + 1$
- Satz: g ist nicht berechenbar.
- Beweis (indirekt): sonst $\exists i$ mit $f_i = g$, betrachte $g(i)$

Eigenschaften von Grammatiken

- E_3 : Sprach-Äquivalenz von Typ-3-Grammatiken
- E_2 : Sprach-Äquivalenz von Typ-2-Grammatiken

gesucht ist jeweils Algorithmus mit dieser Eigenschaft:

Eingabe ist Paar (G_1, G_2) von Grammatiken, Ausgabe ist 1, falls $L(G_1) = L(G_2)$, sonst 0

praktische Motivation: Test bzw. Verkleinerung von regulären Ausdrücken, von Grammatiken (automatische Bewertung von Übungsaufgaben zu AFS!)

- E_3 ist entscheidbar, siehe Vorlesung AFS
- E_2 nicht entscheidbar! diese Vorlesung (aber nicht heute)

Methode: *Reduktion*: wenn E_2 entscheidbar, dann auch ...

Sind diese Aufgaben gleich schwer?

(eine typische Frage der Komplexitätstheorie)

- Def. Eine k -Knoten-Färbung eines Graphen $G = (V, E)$ ist Funktion $f : V \rightarrow \{1, 2, \dots, k\}$ mit $\forall uv \in E : f(u) \neq f(v)$.
- Def. $k\text{COL} :=$ die Menge der Graphen, die eine k -Knoten-Färbung besitzen.
- Probleme:
 - gegeben G , entscheide $G \in 2\text{COL}$
 - gegeben G , entscheide $G \in 3\text{COL}$
- beide Probleme sind entscheidbar (warum?)
- für 2COL ist effizienter Algorith. bekannt, für 3COL nicht.
- Methode: *Reduktion*: wenn man 3COL effizient lösen könnte, dann auch ...

Praktische Problemstellungen

Berechenbarkeitsmodell = Programmierparadigma

- Registermaschine: imperatives Programmieren
- Loop- und While-Programme: strukturiertes (imperat.) P.
- primitiv/allgemein-rekursive Funktionen: funktionales P.
- (uniforme) Schaltkreise: paralleles Programmieren
- nichtdeterministische Maschinen: Suchverfahren

für jede dieser Def.:

- exakte Beschreibung (Spezifikation) von (abstrakter) Syntax und Semantik = Interpreter-Bau

zwischen diesen Def.:

- semantik-erhaltende Übersetzung = Compiler-Bau

Geschichte des Algorithmusbegriffs

Suche nach *Lösungsverfahren* für mathematische Aufgaben (symbolische Differentiation, Integration, Gleichungssysteme)

- Wahrheit einer prädikatenlogischen Formel
- das 10. Hilbertsche Problem (1900):
Lösbarkeit von Polynomgleichungen in ganzen Zahlen

... bzw. nach *Beweisen für deren Nicht-Existenz*

- Gödel, Church, Turing (1936,...):
... ist nicht entscheidbar
- Matjasevich (1970):
... ist nicht entscheidbar.

Bedeutung des Algorithmenbegriffs

(nach K. Wagner: Theor. Inf., Springer 2003)

- Die Bedeutung des Algorithmenbegriffs für Mathematik und Informatik entspricht der Bedeutung des Begriffes der natürlichen Zahlen.
- Die mathematische Präzisierung des Algorithmenbegriffs und die Erkenntnis der Grenzen des algorithmisch Machbaren gehören zu den wichtigsten intellektuellen Leistungen des 20. Jahrhunderts.

Literatur

(akt.) Lehrbücher

- Juraj Hromkovic: *Algorithmische Konzepte der Informatik* Teubner 2001
- Klaus Wagner: *Theoretische Informatik* Springer 2003
- Ingo Wegener: *Theoretische Informatik* Teubner 1992

Klassisch:

- Hartley Rogers Jr.: *Theory of Recursive Functions and Effective Computability*, 1987
- Michael Garey, David S. Johnson: *Computers and Intractability* Freeman 1979

2 Registermaschinen

Motivation, Eigenschaften

wir formalisieren das *imperative* Programmieren:

- Programmtext ist Folge von Befehlen
- Programmausführung ist Folge von Zustandsänderungen
- Zustand: Speicherbelegung und Befehlsnummer

Eigenschaften dieses Modells:

- ist einfach in Hardware realisierbar (seit Jahrzehnten werden Rechner so gebaut)
- ist softwaretechnisch unzweckmäßig (der Beweis für die Korrektheit eines Programms sieht ganz anders aus als das Programm selbst)

Semantik: Speicher

- Speicher der Maschine besteht aus Registern (Zellen),
- Registerinhalte sind aus \mathbb{N}
- Register sind numeriert durch \mathbb{N}
- es werden nur endlich viele Register benutzt
- die Menge der möglichen Speicherbelegungen ist
 $S := \{s \mid s \in (\mathbb{N} \rightarrow \mathbb{N}), \{x \mid s(x) \neq 0\} \text{ ist endlich}\}.$
- Bsp. $s(0) = 42, s(1) = 10, \forall x : x \geq 2 \Rightarrow s(x) = 0.$
- Notation für Speicher-Änderungen: $s[x := y]$
ist die Funktion $z \mapsto (\text{if } z = x \text{ then } y \text{ else } s(z)).$
- Bsp: $s[1 := 8](1) = \dots, s[1 := 8](2) = \dots$
- Ü: gilt $s[a := b][c := d] = s[c := d][a := b]$?

Syntax: Befehle und Programme

- Menge B der *Befehle*:
 $\text{Inc } (\mathbb{N}), \text{Dec } \mathbb{N}, \text{Goto } (\mathbb{N}), \text{GotoZ } (\mathbb{N} \times \mathbb{N}), \text{Stop}.$
- Menge P der *Programme* = B^* (Folge von Befehlen)

Bsp. für ein Programm:

[GotoZ 1 5, Dec 1, Inc 0, Inc 0, Goto 0, Stop]

Semantik: Befehle

- Konfigurationsmenge $C \subset \mathbb{N} \times S$
erste Komponente ist Befehlszähler (enthält die Nr. des nächsten auszuführenden Befehls)

- Übergangsrelation des Programms p ist $\text{step}_p \subseteq C \times C$
 $((l, s), (l', s')) \in \text{step}_p$, falls $l < |p|$ und ...
 - wenn $p_l = \text{Inc}(i)$, dann $l' = l + 1, s' = s[i := s(i) + 1]$
 - wenn $p_l = \text{Dec}(i)$, dann ...
 - wenn $p_l = \text{Goto}(k)$, dann ...
 - wenn $p_l = \text{GotoZ}(i, k)$, dann:
 wenn $s(i) = 0$, dann ... sonst ...

Satz: Die Relation step_p ist eine partielle Funktion.

Semantik: Programme

- initiale Konfigur. $I(x)$ mit Eingabe $x = (x_1, \dots, x_n) \in \mathbb{N}^n$:
 ist $(0, s)$ mit $s(1) = x_1, \dots, s(n) = x_n, s(i) = 0$ sonst
- finale Konfiguration
 (l, s) , wobei $l < |p| \wedge p_l = \text{Stop}$
- die Ausgabe $O(l, s)$ einer Konfiguration ist $s(0)$
- Programm P berechnet die partielle Funktion $f : \mathbb{N}^n \leftrightarrow \mathbb{N}$.
 Es gilt $(x, y) \in f$ gdw.
 - $(I(x), F) \in \text{step}_p^*$ und F ist final und $y = O(F)$
- Jede solche part. Fkt. nennen wir Goto-berechenbar
- Die Menge dieser Fkt. nennen wir GOTO

Ü: ein p , das die Funktion $b(x_1) = 42$ berechnet? Ü: die Semantik des leeren Programms (mit $|p| = 0$) ist?

Ein Programm für $x \mapsto 2x$

[GotoZ 1 5, Dec 1, Inc 0, Inc 0, Goto 0, Stop]

wirklich? glauben wir das? nein. wir beweisen:

- das Programm *hält* für jede Eingabe (vgl. Def. step_p^*)

- die Ausgabe ist *korrekt*

wir ordnen jeder Konfiguration (l, s) zu:

- die *Invariante* $s(0) + 2 \cdot s(1)$
- die *Schranke* $s(1)$

und zeigen (für die Teilfolge aller Konfig. mit $l = 0$):

- die Invariante ist: 1. anfangs wahr, 2. invariant, 3. schließlich nützlich.
- die Schranke nimmt ab (um wieviel?) und bleibt ≥ 0 .

Elementare goto-berechenbare Fkt.

diese Funktionen sind goto-berechenbar:

- jede konstante Funktion
- die identische Funktion
- jede Projektion $(x_1, \dots, x_n) \mapsto x_k$
- die Addition
- die schwache Subtraktion $(x_1, x_2) \mapsto \max(0, x_1 - x_2)$, Notation: $x_1 \dot{-} x_2$

Abschluß-Eigenschaften

- wenn $f : \mathbb{N} \rightarrow \mathbb{N}$ und $g : \mathbb{N} \rightarrow \mathbb{N}$ goto-berechenbar sind,
dann auch $x \mapsto f(g(x))$.

Beweis:

Programm für g ; $R_1 := R_0$; $R_0 := 0$; Programm für f .

Ü: warum $R_0 := 0$?

- wenn $f : \mathbb{N}^2 \rightarrow \mathbb{N}$, $g_1, g_2 : \mathbb{N} \rightarrow \mathbb{N}$ goto-berechenbar sind,
dann auch $x \mapsto f(g_1(x), g_2(x))$.
einfach Programm für g_1 , Programm für g_2, \dots ? Nein.

Zusammenfassung GOTO (bis jetzt)

- formalisiert maschinennahe imperative Programmierung
(Programmausführung als Folge von Speicherzustandsänderungen)
- der Befehlssatz ist klein, die Ausdruckskraft scheint gering, aber immerhin ...
- gewisse einfache Funktionen sind in GOTO
- GOTO ist abgeschlossen bzgl. gewisser Operatoren

später werden wir sehen

- die Ausdruckskraft ist tatsächlich sehr hoch,
GOTO = Java-berechenbar = ...

Übungsaufgaben

1. GOTO-Programme für elementare Funktionen: in Olat/Autotool ausprobieren. (Ggf. Highscore-Wertung für kurze Programme.)
2. Ü: gilt $s[a := b][c := d] = s[c := d][a := b]$?
3. Beweisen Sie: die Relation $\text{step}_p^* \cap \{(C_1, C_2) \mid C_2 \text{ ist final}\}$ ist eine partielle Funktion.
(Dabei Wdhlg. Begriffe und Notation für Relationen und partielle Funktionen.)
4. Sei A die Menge der partiellen Fkt., die durch ein Goto-Programm berechnet werden können, in dem der Befehl Goto nicht vorkommt. Beweisen Sie $\text{GOTO} = A$.
Das sind zwei Inklusionen, die eine ist trivial, für die andere übersetzen Sie einen unbedingten in einen bedingten Sprung.
5. Sei B die Menge der partiellen Fkt., die durch ein Goto-Programm berechnet werden können, in dem der Befehl GotoZ nicht vorkommt (m.a.W., nur unbedingte Sprünge),
Geben Sie ein Verfahren an, das entscheidet, ob ein B -Programm eine totale Funktion berechnet.

6. Sei C die Menge der partielle Ftk., die durch ein Goto-Programm berechnet werden können, in dem weder Goto noch GotoZ vorkommen (m.a.W., die Geradeaus-Programme).

Welche Geradeaus-Programme berechnen totale Funktionen?

Beweisen Sie $B = C$. (zwei Inklusionen, d.h. zwei Compiler)

3 Strukturierte Programmierung

Motivation

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

Struktur = Hierarchie = Bäume.

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

NB: Das ist immer noch imperative Programmierung, deswegen immer noch schlecht für den Programmierer (weil die Semantikdefinition einen Maschinenzustand benutzt, den man im Programm nicht sieht).

Ausweg: funktionale Programmierung (kein Zustand).

Syntax

Menge der While-Programme P

- elementare: $\text{Inc } \mathbb{N}$, $\text{Dec } \mathbb{N}$, leeres Programm: Skip
- zusammengesetzte:
 - Nacheinander: $\text{Seq}(P \times P)$
 - Verzweigung: $\text{IfZ}(\mathbb{N} \times P \times P)$
 - Schleife: $\text{While}(\mathbb{N} \times P)$
- (kein Stop, kein Goto)

Beispiel:

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

Semantik (Prinzip, elementare Prog.)

Semantik eines Programms $p \in P$

ist Relation (genauer: partielle Funktion) $\text{sem}_p \subseteq S \times S$ auf Speicherbelegungen.
das ist *big step semantics* (ein Schritt!)

beachte: es gibt keinen *program counter*, diese Rolle übernimmt der Index p .

Semantik für elementare Programme: $\text{sem}_p(s_1, s_2) =$

- $p = \text{Skip} \wedge s_1 = s_2$
- oder $p = \text{Inc}(i) \wedge s_2 = s_1[i := s_1(i) + 1]$
- oder $p = \text{Dec}(i) \wedge s_2 = s_1[i := \max(0, s_1(i) - 1)]$
- oder ... (nächste Folie)

Semantik für zusammengesetzte Prog.

$\text{sem}_p(s_1, s_2) = \dots$

- oder $p = \text{Seq}(p_1, p_2) \wedge \exists s' : \text{sem}_{p_1}(s_1, s') \wedge \text{sem}_{p_2}(s', s_2)$.
- oder $p = \text{IfZ}(i, p_1, p_2) \wedge$
 $(s_1(i) = 0 \wedge \text{sem}_{p_1}(s_1, s_2) \text{ oder } s_1(i) > 0 \wedge \text{sem}_{p_2}(s_1, s_2))$
- oder $p = \text{While}(i, q) \wedge$
 $(s_1(i) = 0 \wedge s_1 = s_2 \text{ oder } s_1(i) > 0 \wedge \text{sem}_{\text{Seq}(q,p)}(s_1, s_2))$

Notation $p \cong q \iff \text{sem}_p = \text{sem}_q$. — Ü: Satz

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

Ü: IfZ wird gar nicht benötigt, da man es simulieren kann.

While-berechenbare Funktionen

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

Übungsaufgaben:

- die üblichen elementaren Funktionen sind $\in \text{WHILE}$
- **WHILE** ist abgeschlossen unter Substitution
Bsp: $f, g \in \text{WHILE} \Rightarrow (x \mapsto f(g(x))) \in \text{WHILE}$

Ein Interpreter für WHILE

Interpreter realisiert Semantik. Echter autotool-Quelltext: <https://gitlab.imn.htwk-leipzig.de/autotool/all0/blob/master/collection/src/While/Step.hs>

Konfiguration $c = (t, m)$ enthält

- t (todo): Liste (Keller) von Programmen
- m (memory): Speicherbelegung

mit der Bedeutung: wenn $t = [p_1, \dots, p_n]$, dann ist $S(t) := \text{Seq}(p_1, \dots, \text{Seq}(p_n, \text{Skip}) \dots)$ noch auszuführen.

Relation (small-step semantics) \rightarrow auf Konfigurationen.

Spezifikation (Korrektheit): $\text{sem}_{S(t)}(i, f) \iff (|t| = 0 \wedge i = f) \vee (\exists t', m : (t, i) \rightarrow (t', m) \wedge \text{sem}_{S(t')}(m, f))$
es soll gelten: Satz: $\text{sem}_p(i, f) \iff ([p], i) \rightarrow^* ([], f)$

Beziehungen zw. Goto- und While-B.

Satz (Ziel): für jede part. Fkt. f gilt:

- f ist while-berechenbar $\iff f$ ist goto-berechenbar.
- äquivalent, kürzer: **WHILE = GOTO**

praktisches Argument für „ \implies “: das macht jeder Compiler (etwa von C nach Assembler/Maschinensprache)

Beweis (Ideen): folgen.

(Wenn man das genau macht, dann heißt die Vorlesung „Compilerbau“)

Von While zu Goto (Prinzip)

wir schreiben den Übersetzer:

$\text{compile} : \mathbb{N} \times \text{WHILE} \rightarrow \mathbb{N} \times \text{GOTO}$

wobei $\text{compile}(a, p) = (e, q)$ bedeutet:

- das Programm $p \in \text{WHILE}$ wird übersetzt
- in ein äquivalentes Programm $q \in \text{GOTO}$,
- das auf Adresse a beginnt
- und auf $e - 1$ endet (d.h. $e = a + |q|$)

dabei bedeutet „Äquivalenz“:

$\forall p \in \text{WHILE}, a \in \mathbb{N} : \text{sei } (e, q) = \text{compile}(a, p),$

dann $\forall s_1, s_2 : \text{sem}_p(s_1, s_2) \iff \text{step}_q^*((a, s_1), (e, s_2))$

Von While zu 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:

- $\text{compile}(a, \text{Seq}(p_1, p_2)) =$
sei $(m, q_1) = \text{compile}(a, p_1)$ und $(e, q_2) = \text{compile}(m, p_2)$,
dann $(e, q_1 \circ q_2)$.

Von While zu Goto (IfZ)

Ansatz:

$\text{compile } (_, \text{IfZ } i \text{ } p1 \text{ } p2) \Rightarrow$

A: GotoZ i M

compile (_, p2) ;

Goto E ;

M: compile (_, p1) ;

E:

Realisierung:

```
compile (a, IfZ i p1 p2) =
  let (h,q2) = compile (a+1,p2)
      (e,q1) = compile (h+1,p1)
  in  (e, [GotoZ i (h+1)] ++ q2
      ++ [Goto e] ++ q1)
```

Von While zu Goto (While)

Ansatz:

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

Realisierung:

```
compile (a, While i p) =
  let (h,q) = compile (a+1,p)
      e = h+1
  in  (e, [GotoZ i e] ++ q ++ [Goto a])
```

Von While zu Goto (insgesamt)

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

(äquivalente Formulierung: $\text{WHILE} \subseteq \text{GOTO}$)

Beweis(plan):

- $q = \text{compile}(0, p) \circ [\text{Stop}]$
- Aussage folgt aus Korrektheit bzgl. der Spezifikation
 $\forall p \in \text{WHILE}, a \in \mathbb{N} : \text{sei } (e, q) = \text{compile}(a, p),$
dann $\forall s_1, s_2 : \text{sem}_p(s_1, s_2) \iff \text{step}_q^*((a, s_1), (e, s_2))$

https://gitlab.imn.htwk-leipzig.de/autotool/all0/blob/master/collection/src/Compiler/While_Goto.hs

Von Goto zu While

das scheint schwieriger:

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

Es geht aber, und das erzeugte While-programm hat eine ganz besondere (einfache) Struktur, die später noch ausgenutzt wird (Kleene-Normalform-Thm)

Von Goto nach While: Ansatz

Eingabe: goto-Programm p , Ausgabe: äquivalentes While-programm q

bestimme c = das erste in p nicht benutzte Register, das verwenden wir als PC. Das nächste Register 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
      }
    }
  }
}
```

Von Goto nach While: Einzelheiten

für Befehl p_i erzeuge: $\text{if } (c==i) \text{ } q_i \text{ else mit } q_i =$

- wenn $p_i \in \{\text{Inc } r, \text{Dec } r\}$, dann $[p_i, \text{Inc } c]$
- wenn $p_i = \text{Stop}$, dann $[\text{Dec } h]$
- wenn $p_i = \text{Goto}(l)$, dann $[c := l]$,
- wenn $p_i = \text{GotoZ}(r, l)$, dann $\text{IfZ } r \text{ } (c := 1) \text{ } (\text{Inc } c)$

Ü: zeige: p erreicht Stop $\iff q$ hält.

beachte dabei auch den Fall **Goto** l mit $l \geq |p|$

Ü: hier wird $\text{if } (c==i)$ und $c := l$ benutzt, das kann man jeweils mit While implementieren, geht hier aber auch ohne Schleife, warum?

https://gitlab.imn.htwk-leipzig.de/autotool/all0/blob/master/collection/src/Compiler/Goto_While.hs

Das Normalform-Theorem für While

Vorige Konstruktion zeigt den Satz:

- zu jedem Goto-Programm gibt es ein äquivalentes While-programm ($\text{GOTO} \subseteq \text{WHILE}$)
- mit *genau einem* While.

zusammen mit $\text{WHILE} \subseteq \text{GOTO}$ folgt

- $\text{WHILE} = \text{GOTO}$
- zu jedem While-Programm gibt es ein äquivalentes While-Programm mit genau einem While.

„äquivalent“ = berechnet dieselbe partielle Funktion.

Ü: wie unterscheiden sich die Laufzeiten?

4 Garantierte Termination: Loop

Motivation

- $\text{While}(i, q)$ bedeutet: solange $s(i) > 0$ ist, q ausführen
- es gibt While-Programme, die nicht für jede Eingabe terminieren (es gibt $f \in \text{WHILE}$ mit f nicht total)
- $\text{Loop}(i, q)$ bedeutet: q genau $s(i)$ mal ausführen
(der Wert von i vor Beginn der Schleife)
- Loop-Programme terminieren (jede $f \in \text{LOOP}$ ist total)
das ist softwaretechnisch nützlich
- aber auch eine Einschränkung:
es gibt $f \in (\text{WHILE} \cap \text{TOTAL}) \setminus \text{LOOP}$

Loop-Programme

Syntax und Semantik wie While-Programme, außer:

- (Syntax) kein $\text{While}(i, q)$, sondern $\text{Loop}(i, q)$
- (Semantik) wenn $p = \text{Loop}(i, q)$, dann $\text{sem}_p(s_1, s_2) = \text{sem}_q^{s_1(i)}(s_1, s_2)$
der Befehl q wird genau $s_1(i)$ mal ausgeführt (der Wert von i , wenn die Schleife zu
erstmals betreten wird — egal, was später mit i passiert)

Jede so berechenbare Fkt. heißt loop-berechenbar.

Die Menge der loop-berechenbaren Fkt. heißt LOOP.

Bsp: Addition, Subtraktion, Multiplikation, Potenz,

$n \mapsto n$ ist gerade, $n \mapsto n$ ist Quadratzahl, $n \mapsto n$ ist prim, ...

Loop-Programme und Softwaretechnik

- bei $\text{Loop}(i, q)$ wird q genau $s_1(i)$ -mal durchlaufen
- so realisiert in der Sprache Ada (http://www.adaic.org/resources/add_content/standards/12rm/html/RM-5-5.html)
„A loop parameter is a constant; it cannot be updated...“

```
for X in 0 .. 10 loop P; end loop;
```

- Iteration (Induktion) über (Peano-)Zahlen (Strichlisten)
- verallgemeinert auf andere (strukturierte) Datentypen: Rekursionsmuster (`fold`), Entwurfsmuster Iterator
(besucht jedes Element genau einmal)
Java: `for (E x : c) { .. }`, C#: `foreach`

LOOP und WHILE

- $\text{LOOP} \subseteq \text{WHILE} \cap \text{TOTAL}$ (Beweis: Übung)
- $\text{WHILE} \cap \text{TOTAL} \not\subseteq \text{LOOP}$. Beweis:
 - L_0, L_1, \dots längen-lexikografische Aufzählung aller LOOP-Programme, die einstellige Fkt. berechnen,
diese Fkt. sind f_0, f_1, \dots

- für $d : x \mapsto f_x(x) + 1$ gilt: $d \in \text{WHILE} \cap \text{TOTAL}$
Begründung: Interpreter für LOOP-Programme
- dieses d hat keinen L -Index (also $d \notin \text{LOOP}$)
Begründung: falls doch $d = f_i$, dann betrachte $d(i)$.
- eine arithmetische Funktion $f \in \text{WHILE} \cap \text{TOTAL} \setminus \text{LOOP}$: die Ackermann-Funktion

Die Ackermann-Funktion

- $A : \mathbb{N}^2 \rightarrow \mathbb{N}$
 $A(0, y) = y + 1; A(x + 1, 0) = A(x, 1);$
 $A(x + 1, y + 1) = A(x, A(x + 1, y))$
- bestimme $A(2, 4), A(3, 3), A(4, 2)$
- Satz: $\forall f \in \text{LOOP} \exists x : \forall \vec{y} : f(\vec{y}) \leq A(x, \max_i y_i)$
folgt aus
- Lemma: (Bezeichnung $\max s := \max\{s(i) \mid i \in \mathbb{N}\}$)
 $\forall p \in \text{LOOP} \exists x : \forall (s_1, s_2) \in \text{sem}_p : A(x, \max s_1) \geq \max s_2$
Beweis durch Induktion über Programmtexte

Eigenschaften der Ackermann-Funktion

(..., die im Beweis des Lemmas benötigt werden)

- für $p = \text{Inc}(i)$: (Übung)
- für $p = \text{Seq}(p_1, p_2)$ betrachte Zustände $s_1 \xrightarrow{\text{sem}_{p_1}} s' \xrightarrow{\text{sem}_{p_2}} s_2$
und nach Induktion $A(x_1, m_1) \geq m', A(x_2, m') \geq m_2$.
Gesucht ist für jedes x_1, x_2 ein x mit $\forall y : A(x, y) \geq A(x_1, A(x_2, y))$.
Wir können $x = x_1 + x_2 + 2$ wählen
(Beweis: Übung. Benötigt $A(x, y + 1) \leq A(x + 1, y)$)
- für $p = \text{Loop}(i, q)$: Welche Eigenschaft wird benötigt? (Ü)

Übung KW 47

1. Aufgaben zu While- und Loop-Programmen in autotool
2. Sei W_{Syn} die Menge der While-Programme, in denen `lfZ` nicht vorkommt, und W_{Sem} die Menge der durch solche Programme berechenbaren partiellen Funktionen.
Zeigen Sie $W_{\text{Sem}} = \text{WHILE}$.
3. Zeigen Sie für einstellige partielle Funktionen:
 $f, g \in \text{WHILE} \Rightarrow (x \mapsto f(g(x))) \in \text{WHILE}$.
4. Zur Kompilation von Goto nach While:
 - (a) Zeigen Sie: p erreicht Stop $\iff q$ hält
 - (b) Es werden `if (c==i)` und `c := 1` benutzt, das kann man im Allgemeinen mit While implementieren (wie?) geht aber hier auch ohne Schleife (wie?)
 - (c) Vergleichen Sie die Laufzeiten von p und q .
5. zur Ackermann-Funktion:
 - (a) Bestimmen Sie $A(2, 4)$, $A(3, 3)$, $A(4, 2)$
 - (b) Aufgaben auf Folie „Eigensch. Ackermann“

5 Kodierung strukturierter Daten

Motivation

- In vielen Anwendungen sind Daten strukturiert (z.B. Tupel, Listen, Bäume). Goto-Programme rechnen aber nur mit Zahlen.
- Satz: Das ist keine Einschränkung der Allgemeinheit, denn man kann jedes strukturierte Datum in eine einzige (mglw. große) Zahl kodieren.
- wird *Gödelisierung* genannt (Kurt Gödel, 1906–1978)
- anschauliches Argument: der Speicherinhalt eines PC ist eine Bitfolge, die kann man als Binärdarstellung einer Zahl auffassen. — exakte Argumente: folgen.

Kodierung von Zahlenpaaren

gesucht sind (goto-berechenbare) Funktionen

- Konstruktor: $C : \mathbb{N}^2 \rightarrow \mathbb{N}$ Destruktoren: $P_1, P_2 : \mathbb{N} \rightarrow \mathbb{N}$
- Testfunktion: $T : \mathbb{N} \rightarrow \{0, 1\}$

mit Spezifikation (vgl. objektorientierte Datenmodellierung)

- $\forall x_1, x_2 : P_1(C(x_1, x_2)) = x_1 \wedge P_2(C(x_1, x_2)) = x_2$
- $\forall x : T(x) = 1 \iff \exists x_1, x_2 : x = C(x_1, x_2)$

da gibt es viele verschiedene Möglichkeiten

- $C(x_1, x_2) = (x_1 + x_2)(x_1 + x_2 + 1)/2 + x_1$
- $C(x_1, x_2) = 2^{x_1}(2x_2 + 1), \quad \bullet C(x_1, x_2) = 2^{x_1} \cdot 3^{x_2}$

Ü: jeweils $C(2, 3), C(3, 2), T(10), T(12), P_1(12), P_2(12)$,
Algorithmen (Loop-Programme) für T, P_i .

Kodierung von Listen

Konstruktor: $L : \mathbb{N}^* \rightarrow \mathbb{N}$, Destruktoren $D_i : \mathbb{N}^* \leftrightarrow \mathbb{N}$.

zwei (von vielen) Möglichkeiten:

- mittels einer Paar-Kodierung C

```
L (l) = if null l then 0
        else 1 + C (head l, L (tail l))
```

- direkte Kodierung als Produkt von Primzahlpotenzen $L([x_0, x_2, \dots, x_n]) = 2^{x_0+1} \cdot 3^{x_1+1} \cdot \dots \cdot p(n)^{x_n+1}$.

Ü: jeweils $L([], L([5]), L([2, 3, 0]))$, Algorithmus für D_i

Ü: die Funktion $p : n \mapsto$ die n -te Primzahl, also $p(0) = 2, p(1) = 3, p(2) = 5, \dots$ ist

While-berechenbar.

Ü : p ist Loop-berechenbar. — Hinweis: Euklid.

Kodierung von Bäumen

Motivation

- allgemein: Baum = Term in einer Signatur,
- Signatur: eine endlichen Menge von Funktionssymbolen mit zugeordneter Stelligkeit.
- Bsp: $\Sigma = \{(f, 2), (g, 1), (a, 0)\}$, $t = f(f(a, g(a)), a) \in \text{Term}(\Sigma)$.
Notation $\text{root}(t) = f$, $\text{args}(t) = [f(a, g(a)), a]$.
- wird u.a. benötigt, um (prädikatenlogische) Formeln als Zahlen zu kodieren.

Realisierung: $B(t) = C(\text{num}(\text{root}(t)), L([B(t_1), \dots, B(t_k)]))$

mit $\text{args}(t) = [t_1, \dots]$, Paar-Kodierung C , Listen-Kodierung L , sowie Symbol-Numerierung

Ü: Kodierung für endliche Mengen von Zahlen

6 Universelle Programme und Halteproblem

Kodierung von Programmen

man kann mit eben gezeigten Methoden nach \mathbb{N} kodieren:

- Goto-Programme
(Programm ist Liste von Befehlen, Befehl ist Tupel)
- Maschinen-Konfigurationen
(Paar von Zahl und Speicherbelegung, diese ist Liste (!))

Damit kann man in der Sprache GOTO einen *Interpreter* für GOTO-Programme schreiben.

Ein universelles Goto-Programm

- Insbesondere sind für jedes p die Übergangsfunktion step_p sowie ihre transitive reflexive Hülle goto-berechenbar (nach Kodierung):
- bei Eingabe einer Kodierung von p und einer Konfig. K kann die Folgekonfiguration berechnet werden und dies solange wiederholt werden, bis finale Konf. erreicht wird.

- D.h. die part. Funktion $\phi : \mathbb{N}^2 \hookrightarrow \mathbb{N}$ ist goto-berechenbar: $\phi_x(y) =$ die Ausgabe einer Maschine, die das Programm mit Kodierung x auf Eingabe mit Kodierung y ausführt.
- Das Programm für ϕ heißt *universell*, denn es kann die Rechnung *jedes* Goto-Programms simulieren.

Das Halteproblem

Def: das (spezielle) Halteproblem ist die Menge $K_0 = \{x \mid \phi_x(x) \downarrow\} \subseteq \mathbb{N}$.

(die Menge der Kodierungen von Programmen, die anhalten, wenn man sie auf „sich selbst“, d.h. ihren eigenen Code, anwendet).

Satz: die charakteristische Funktion $c_{K_0} : \mathbb{N} \rightarrow \{0, 1\}$ der Menge K_0 ist nicht goto-berechenbar.

Beweis (indirekt): falls doch, dann gibt es ein Programm, das c_{K_0} berechnet. Es gibt dann auch ein Programm

$x \mapsto$ wenn $c_{K_0}(x) = 0$, dann 1, sonst \perp (eine nicht haltende Rechnung).

Sei q der Code dieses Programms. Ist $q \in K_0$? Gdw. $\phi_q(q) \downarrow$, gdw. $c_{K_0}(q) = 0$, gdw. $q \notin K_0$.

Das Halteproblem (Folgerung)

- Satz: es gibt Funktionen $\mathbb{N} \rightarrow \mathbb{N}$, die durch kein goto-Programm berechenbar sind.
Beweise: c_{K_0}
- Def: das (allgemeine) Halteproblem ist die Menge $K = \{C(x, y) \mid \phi_x(y) \downarrow\}$.
- Satz: charakterist. Funkt. c_K ist nicht goto-berechenbar.
- Beweis: sonst könnte man auch c_{K_0} berechnen.

Es gibt also kein allgemeines Verfahren, mit dem man entscheiden kann, ob ein Programm für eine Eingabe nach endlich vielen Schritten hält.

Diagonalisierung

schon zweimal benutzt, und kommt noch öfter:

- für eine Menge F von Funktionen gibt es eine Aufzählung $f_0, f_1, f_2 \dots$
- konstruiere $g : x \mapsto$ geeignete Änderung von $f_x(x)$,
so daß g in Aufzählung nicht vorkommt ($\neg \exists i : g = f_i$)

Anwendungen bisher:

- F = die totalen berechenbaren Funktionen
⇒ es gibt totale nicht berechenbare Fkt.
- F = die partiellen berechenbaren Funktionen
⇒ das Halteproblem ist nicht entscheidbar

7 Maschinenunabhängige Berechenbarkeitstheorie

Motivation

- wir haben gezeigt: GOTO = WHILE
die Syntax und Semantik (Interpreter) waren jeweils spezifisch, aber wir haben Compiler konstruiert
- Verallgemeinerung (Alonzo Church, Alan Turing)
alle vernünftigen Berechenbarkeitsmodelle definieren die gleiche Klasse von partiellen Funktionen
- weitere Modelle: Wortersetzung (Turing-Maschinen), funktionale Programmierung (rekursive Fkt.)

Übliche Namen für Funktionenklassen

- $f : \mathbb{N}^* \leftrightarrow \mathbb{N}$ ist *partiell-rekursiv*, $f \in \text{Part}$:
 f ist While-berechenbar (= Goto-berechenbar
= Turing-berechenbar = ...-berechenbar = ...)
- $f : \mathbb{N}^* \leftrightarrow \mathbb{N}$ ist *allgemein-rekursiv*, $f \in \text{Allg}$:
 f ist partiell rekursiv *und total*.
- $f : \mathbb{N}^* \rightarrow \mathbb{N}$ ist *primitiv-rekursiv*, $f \in \text{Prim}$:
 f ist Loop-berechenbar (= ...)

Begründung:

- While-Schleife \iff beliebige Rekursion
- Loop-Schleife \iff eingeschränkte (primitive) Rekursion

These von Church und Turing

- „alle intuitiv vernünftigen Berechenbarkeitsmodelle definieren die gleiche Klasse von partiellen Funktionen“
- ein Berechnungsmodell ist gegeben durch
 - eine Tupel-Kodierung C
 - eine universelle Funktion (Interpreter) $\phi : \mathbb{N} \times \mathbb{N} \leftrightarrow \mathbb{N}$und bestimmt Menge von in diesem Modell berechenbaren partiellen Funktionen $M = \{\vec{x} \mapsto \phi_p(C(\vec{x})) \mid p \in \mathbb{N}\} \subseteq (\mathbb{N}^* \leftrightarrow \mathbb{N})$
- Die C-T-These kann man auffassen als empirische Aussage oder als Definition (M ist vernünftig \iff es gibt beide Compiler $M \leftrightarrow \text{WHILE}$)

Ein Fixpunktsatz

Satz (Stephen Kleene, 1938): Sei f total und berechenbar. Dann gibt es ein i mit $\phi_i = \phi_{f(i)}$.

Beweis:

- bestimme h , so daß $h(x)$ ein Index für diese Funktion ist: $y \mapsto \phi_{\phi_x(x)}(y)$.
- Bestimme e als einen Index für $x \mapsto f(h(x))$.
- Das gesuchte i ist $h(e)$.

Ü: wende Satz an auf die Funktion $f : x \mapsto$ ein Index für die konstante Funktion $y \mapsto x$.

Der Fixpunkt-Index für f ist (indiziert) ein Programm, das seinen eigenen Quelltext ausgibt.

Geht in *jeder* (in unserem Sinne vernünftigen) Sprache!

Der Satz von Rice

Satz: jede nichttriviale semantische Eigenschaft von Programmen ist unentscheidbar. dabei bedeuten:

- Eigenschaft: Menge $E \subseteq \mathbb{N}$ von Gödelnummern
- nichttrivial: $E \neq \emptyset \wedge E \neq \mathbb{N}$
- semantisch: $\forall x, y : (\phi_x = \phi_y) \Rightarrow (x \in E \iff y \in E)$

Beispiele (semantisch oder nicht?)

- das Programm berechnet eine totale Funktion
- $\{x \mid \text{dom}(\phi_x) = \mathbb{N}\}$
- die Länge des Programmtextes ist eine gerade Zahl
- $\{x \mid \text{dom}(\phi_x) = 2\mathbb{N}\}$
- die Gödelnummer ist gerade ($2\mathbb{N}$)

Der Satz von Rice (Beweis)

- wähle $y \in E, n \in \mathbb{N} \setminus E$.
- sei E entscheidbar, d.h., c_E berechenbar.
Dann ist dieses f berechenbar und total:
 $f : x \mapsto \text{wenn } c_E(x) = 1, \text{ dann } n, \text{ sonst } y$.
- Nach Konstruktion $x \in E \iff f(x) \notin E$.
- nach Fixpunktsatz gibt es x mit $\phi_x = \phi_{f(x)}$.
- Damit $x \in E \iff f(x) \in E$.

Busy-Beaver-Programme

- vgl. Aufgabe autotool und Übung 5 zu B_{While} .
- ein Programm, das ziemlich lange rechnet:

```
Seq (Inc 1)
  (While 1
    (Seq (Inc 1)
      (Seq (Inc 1)
        (Seq (Inc 1)
          (Seq (While 2
            (Seq (Dec 2)
              (While 1
                (Seq (Inc 2)
                  (Seq (Inc 2) (Dec 1))))))
            (Inc 2))))))
    (Inc 2))))))
```

- für Turingmaschinen:
 - Heiner Marxen, Jürgen Buntrock, *Attacking the Busy Beaver 5*, Bulletin of the EATCS, Number 40, February 1990, pp. 247-251 <https://www.drb.insel.de/~heiner/BB/>
 - Pascal Michel: *Historical Survey of Busy Beavers* <http://www.logique.jussieu.fr/~michel/ha.html>

Übung KW48

1. Beispiel-Rechnungen (siehe Folien) zu Kodierung von Paaren, Listen, Bäumen
2. für die Primzahlfunktion p gilt: $p \in \text{LOOP}$.
Hinweis: $p \in \text{WHILE}$ ist einfach, man muß jetzt zusätzlich eine Loop-berechenbare obere Schranke für $p(n)$ angeben. Diese kann großzügig sein, z.B. aus Beweis von Euklid für „es gibt unendlich viele Primzahlen“.
3. Def. $T(x, y, z) :=$ die vom Goto-Programm x bei Eingabe y nach z Schritten erreichte Konfiguration.
(genauer: x ist die Kodierung des Programmtextes, y ist die Kodierung des Eingavektors, Ausgabe ist die Kodierung einer Konfiguration oder einer Fehlermeldung, falls Programm schon vorher gehalten hat)
 T ist Loop-berechenbar.
4. zur Diagonalisierung: wende das Verfahren an auf
 - $F =$ alle linearen Funktionen $x \mapsto ax + b$ mit $a, b \in \mathbb{N}$, $g(x) = f_x(x) + 1$
 - $F =$ alle Loop-berechenbaren Funktionen, $g(x) = (f_x(x) + 1) \bmod 2$
 - $F = (\mathbb{N} \rightarrow \mathbb{N})$, d.h., alle Funktionen (egal, ob berechenbar), $g(x) = f_x(x) + 1$.
5. Def. $B_{\text{While}}(x) :=$ die größte Schrittzahl aller bei leerer Eingabe haltenden While-Programme der Größe $\leq x$ (vgl. autotool-Aufgabe).
Beweisen Sie: B_{While} ist nicht berechenbar.
Hinweis: indirekt. Wenn B_{While} berechenbar, dann Halteproblem entscheidbar.
6. Entspr. B_{Loop} . Ist $B_{\text{Loop}} \in \text{WHILE}$? Ist $B_{\text{Loop}} \in \text{LOOP}$?
(Ja. Nein. Hinweis: betrachte Loop-Programm für $f : x \mapsto 1 + 2 \cdot B_{\text{Loop}}(x)$, rufe dieses geeignet auf.)

7. Ü-Aufgabe von Folie „Fixpunktsatz“.
8. Geben Sie ein Programm in Java (C, Haskell,...) an, das seinen eigenen Quelltext ausgibt.
(Nur Schreiben auf Standardausgabe, keine Dateioperationen, d.h., Programm darf seinen Quelltext nicht von externem Speicher holen, sondern muß ihn selbst enthalten oder erzeugen)
Hinweis: (Garry Thompson, 1999) <http://www.nyx.net/~gthomps/quine.htm>
9. Beweisen Sie: für jedes $i \in \mathbb{N}$ gibt es unendlich viele Indizes (die Menge $E_i := \{j \mid \phi_i = \phi_j\}$ ist unendlich)
Hinweis: $j \in E_i$ ist eine semantische Eigenschaft.

8 Entscheidbare Mengen, Reduktion

Motivation, Definition REC

- allgemeines Ziel ist Einordnung der Schwierigkeit von Entscheidungsproblemen
- nach Kodierung: Problem = Teilmenge P von \mathbb{N}
zu entscheiden ist dann, ob $x \in P$, d.h. $c_P(x) = 1$
- Def: $\text{REC} = \{P \mid P \subseteq \mathbb{N}, c_P \text{ ist berechenbar}\}$
(Name kommt von „berechenbar durch *rekursive* Fkt.“)
- Beispiele: Primzahlen $\in \text{REC}$, $K_0 \notin \text{REC}$
- konkrete Ziele:
 - (Abschluß-)Eigenschaften von REC
 - Beweisverfahren für $P \in \text{REC}$, $P \notin \text{REC}$
 - genauere Struktur für $2^{\mathbb{N}} \setminus \text{REC}$

Abschlußeigenschaften von REC

Satz (REC ist abgeschlossen unter Booleschen Operationen):
wenn $A, B \in \text{REC}$, dann

- $A \cup B \in \text{REC}$
- $A \cap B \in \text{REC}$

- $(\mathbb{N} \setminus A) \in \text{REC}$

Beweis (Beispiel):

Wenn c_A, c_B rekursive Fkt, dann auch $c_{A \cup B}$: $c_{A \cup B}(x) = \max(c_A(x), c_B(x))$, d.h. $c_{A \cup B} = \text{SUBST}(\max, c_A, c_B)$.

Reduktion \leq_m

Zum Vergleich der algorithmischen Schwierigkeit von Probleme definiert man:

$P \leq_m Q$ („ P ist reduzierbar auf Q “) durch:

es existiert eine berechenbare totale Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ mit $\forall x \in \mathbb{N} : x \in P \iff f(x) \in Q$.

- beachte die Richtung: P ist höchstens so schwierig wie Q
- „reduzieren“ bedeutet: ein Entscheidungsverfahren für P auf ein Verfahren für Q zurückführen.
- Index m kommt von „many-one“-Reduktion

Satz: $P \leq_m Q \wedge Q \in \text{REC} \Rightarrow P \in \text{REC}$.

Beweis: gegeben c_Q , konstruiere $c_P = \text{SUBST}(\dots)$

Satz (Ü): \leq_m ist transitiv

Anwendungen der Reduktion

Def (Wdhlg)

- das allgemeine Halteproblem, $K = \{C(x, y) \mid \phi_x(y) \downarrow\}$.
- das spezielle Halteproblem, $K_0 = \{x \mid \phi_x(x) \downarrow\}$.

Satz: $K_0 \leq_m K$. Beweis: $x \in K_0 \iff C(x, x) \in K$.

Folgerung: wir hatten gezeigt $K_0 \notin \text{REC}$, also gilt $K \notin \text{REC}$.

Ü: zeige $K \leq_m K_0$.

Wir bezeichnen $T = \{x \mid \phi_x \text{ ist total}\}$.

Satz: $T \notin \text{REC}$.

Beweis (Ü): zeige $K \leq_m T$.

Betrachte dazu die 3-stellige (!) Fkt $g : (x, y, z) \mapsto \phi_x(y)$ und wende s_1^2 an.

9 Aufzählbare Mengen

Motivation, Definition RE

$K_0 \notin \text{REC}, T \notin \text{REC}$. Sind beide Probleme gleich schwer?

Nein. K_0 ist rekursiv aufzählbar, T ist es nicht.

Def. $P \subseteq \mathbb{N}$ heißt rekursiv aufzählbar, falls $P = \emptyset$ oder

- es gibt totale berechenbare Funktion f mit $P = f(\mathbb{N})$.

Im zweiten Fall ist $P = \{f(0), f(1), f(2), \dots\}$. — Beachte:

- jedes Element von P kommt wenigstens einmal vor,
- f ist nicht notwendig injektiv (wiederholungsfrei),
- f ist nicht notwendig monoton.

Die Menge der rek. aufzählbaren Mengen heißt RE.

Ü: $P \in \text{RE} \wedge P$ unendlich $\Rightarrow P$ ist injektiv aufzählbar

Ü: P unendlich $\wedge P$ streng monoton aufzählbar $\Rightarrow P \in \text{REC}$.

Eine äquivalente Charakterisierung von RE

(Wdhlg.) $P \in \text{RE} : P = \emptyset$ oder $\exists f \in \text{Allg} : P = \text{rng}(f)$

Satz: $P \in \text{RE} \iff \exists f \in \text{Part} : P = \text{dom}(f)$

Beweis: „ \Rightarrow “. $P = \emptyset$: f hält niemals.

P wird aufgezählt durch $g: f(x) :=$ wenn x in $g(0), g(1), \dots$ vorkommt, dann 1. (sonst hält die Rechnung nicht.)

Beweis: „ \Leftarrow “ trivial für P endlich.

Tabelle mit (x, y) : die Konfiguration nach y Schritten in der Rechnung $f(x)$ (oder Markierung, daß schon fertig).

Tabelle gemäß Kodierung von \mathbb{N}^2 durchlaufen.

Wenn Konfiguration (x, y) final, dann x ausgeben.

Anwendung: $K_0 \in \text{RE}$. Beweis: $K_0 = \text{dom}(x \mapsto \phi_x(x))$.

Bezeichnung (Gödelisierung for RE) $W_x := \text{dom}(\phi_x)$

Abschlußeigenschaften von RE

Satz (offensichtlich): $A \in \text{RE}, B \in \text{RE} \Rightarrow (A \cup B) \in \text{RE}$.

Beweis: trivial wenn $A = \emptyset$ oder $B = \emptyset$.

Sei f die aufzählende Funktion für A , g die für B .

Dann $h(2n) = f(n), h(2n + 1) = g(n)$.

Satz (nicht offensichtlich): $A \in \text{RE}, B \in \text{RE} \Rightarrow (A \cap B) \in \text{RE}$.

Die Schwierigkeit ist: wenn man ein $x = f(n) \in A$ hat, kann man nicht ausrechnen, ob $x \in B$, denn möglicherweise ist $B \notin \text{REC}$.

Beweis: 2-dim. unendliche Tabelle,

In Zeile x , Spalte y steht Zahlenpaar $(f(x), g(y))$.

Wenn ..., gibt ... aus.

Ü: alternativer Beweis über Def.-Bereiche

Eine Beziehung zw. RE und REC

Satz: $P \in \text{RE} \wedge (\mathbb{N} \setminus P) \in \text{RE} \iff P \in \text{REC}$

Beweis: \Leftarrow als Ü. – Für \Rightarrow :

trivial, wenn $P = \emptyset$ oder $P = \mathbb{N}$. — ansonsten:

- Sei f eine Aufzählung für P , g eine Aufzählung für $\mathbb{N} \setminus P$.
- um zu bestimmen, ob $x \in P$:
 - berechne $f(0), g(0), f(1), g(1), f(2), \dots$
 - bis x erscheint
- diese Verfahren hält. (Beweis durch Fallunterscheidung.)

Diese Aussage in anderer Bezeichnung:

Def: $\text{coRE} = \{P \mid (\mathbb{N} \setminus P) \in \text{RE}\}$ Satz: $\text{RE} \cap \text{coRE} = \text{REC}$.

Folgerung: $(\mathbb{N} \setminus K_0) \notin \text{RE}$.

RE und \leq_m

- Satz: $P \leq_m Q \wedge Q \in \text{RE} \Rightarrow P \in \text{RE}$.
- Beweis (Variante 1)
 f die Funktion aus der Reduktion \leq_m , $Q = \text{dom}(\phi_i)$.
Dann $P = \text{dom}(x \mapsto \phi_i(f(x)))$.
- Beweis (Variante 2 – Übung)
Sei g die Aufzählung für Q .
(Falls existiert. Sonst $Q = \emptyset$, was dann?)
Bestimme Aufzählung für P mit 2-dim unendl. Tabelle.
Was steht in Spalte x , Zeile y , was wird ausgegeben?

Ü: $K_0 \not\leq_m (\mathbb{N} \setminus K_0)$, $(\mathbb{N} \setminus K_0) \not\leq_m K_0$

Die schwersten Probleme in RE

Def: Q heißt RE-vollständig (bzgl. \leq_m), falls:

- $Q \in \text{RE}$ und $\forall P \in \text{RE} : P \leq_m Q$.

Satz: K ist RE-vollständig.

Beweis: Sei $P \in \text{RE}$, gegeben als $\text{dom}(\phi_i)$.

Dann $x \in P \iff x \in \text{dom}(\phi_i) \iff \phi_i(x) \downarrow \iff C(i, x) \in K$.

Die Reduktion (\leq_m) benutzt also die Funktion $x \mapsto C(i, x)$.

Übungen:

- P ist RE-vollständig $\Rightarrow P \notin \text{REC}$.
- P ist RE-vollst. $\wedge P \leq_m Q \wedge Q \in \text{RE} \Rightarrow Q$ ist RE-vollst.
- definiere den analogen Begriff „REC-vollständig“. Welche analogen Sätze gelten?
Welche Mengen sind REC-vollst? (Zu viele, deswegen ist das nicht interessant)

Übungsaufgaben

Aufgaben für Übung KW49/KW50 sind markiert.

Aufgaben mit (!) enthalten evtl. schwere Teilaufgaben. Bilden Sie sich trotzdem eine Meinung.

1. (KW49) \leq_m ist: transitiv, reflexiv?, symmetrisch? antisymmetrisch?

2. (KW 49) $K \leq_m K_0$.

Musterlösung: die Reduktionsfunktion f bildet Eingabe x ab auf einen Index der Funktion $z \mapsto \phi_{P_1(x)}(P_2(x))$. Dann $x \in K \iff \phi_{P_1(x)}(P_2(x)) \downarrow \iff \phi_{f(x)}(f(x)) \downarrow \iff f(x) \in K_0$.

3. (KW 49) (!) gehören diese Mengen zu REC, RE, coRE?

$\{x \mid W_x = \emptyset\}, \{x \mid 1 = |W_x|\}, \{x \mid W_x \text{ ist endlich}\}, \{x \mid W_x \text{ ist unendlich}\}, \{x \mid W_x = \mathbb{N}\}, \{C(x, y) \mid W_x = W_y\}$,

dabei ist C eine Paar-Kodierung.

4. (KW 49) Definition: $A \otimes B := \{C(x, y) \mid x \in A, y \in B\}$.

Für A, B in REC bzw. RE (d.h., 4 Fälle): ist $A \otimes B$ in REC bzw. RE? (d.h., je 2 Fragen)

Musterlösung (teilw.)

- Es gilt $\forall A \in \text{REC}, B \in \text{REC} : A \otimes B \in \text{REC}$, denn $c_{A \otimes B}(x) = c_A(P_1(x)) \wedge c_B(P_2(x))$. Nach Voraussetzung sind c_A und c_B allgemein rekursiv, also auch $c_{A \otimes B}$.
- Es gilt nicht $\forall A \in \text{RE}, B \in \text{RE} : A \otimes B \in \text{REC}$. Gegenbeispiel: $A = \mathbb{N}, B = K_0$. Dann gilt $K_0 \leq_m \mathbb{N} \otimes K_0$ mit Reduktionsfunktion $f_x \mapsto C(42, x)$. Aus $K_0 \notin \text{REC}$ folgt $\mathbb{N} \otimes K_0 \notin \text{REC}$.
- Es gibt Mengen $A \in \text{RE}, B \in \text{RE} \setminus \text{REC}$ mit $A \otimes B \in \text{REC}$. Beispiel: $A = \emptyset, B = K_0$, denn $\emptyset \otimes K_0 = \emptyset \in \text{REC}$

5. gibt es für jede $A, B \in \text{RE}$ mit $A \cap B = \emptyset$

- $f \in \text{Part}$ mit $f(A) = \{0\}$ und $f(B) = \{1\}$?
- $f \in \text{Allg}$ mit $f(A) = \{0\}$ und $f(B) = \{1\}$?

6. $f : \mathbb{N} \rightarrow \mathbb{N}$ heißt *Permutation*, wenn f bijektiv ist.

Zeige, daß die (allgemein) rekursiven Permutationen bzgl. Nacheinanderausführung eine Gruppe bilden

(... aber die primitiv rekursiven nicht)

7. (KW 49) Jede unendliche rek. aufzählbare Menge hat eine unendliche entscheidbare Teilmenge.

aber (!) es gibt eine unendliche Menge ohne unendliche rek. aufzählbare Teilmenge.

8. (KW 50)

- $\{x \mid W_x \text{ ist unendlich}\} \equiv_m \{x \mid W_x = \mathbb{N}\}$

Musterlösung: bezeichne die linke Seite mit A , rechte mit B .

- Zu zeigen: $A \leq_m B$. Die Reduktionsfunktion f bildet x ab auf einen Index des folgenden Programms:

$e \mapsto$ bei der Diagonal-Durchquerung der Tabelle von „ $(a, b) \mapsto \phi_x(a)$ “ hält nach genau b Schritten“ wird wenigstens e mal True angetroffen.

Dann $x \in A \iff$ Tabelle enthält unendlich viele True $\iff \phi_{f(x)}$ ist total $\iff f(x) \in B$.

- zu zeigen: $B \leq_m A$. Die Reduktionsfunktion f bildet x ab auf einen Index des folgenden Programms:

$e \mapsto$ for i from 0 to $e-1$ do $\phi_x(i)$.

Dann $x \in B \Rightarrow \phi_x$ total $\Rightarrow \phi_{f(x)}$ total

und $x \notin B \Rightarrow$ es gibt ein kleinstes i mit $\phi_x(i) \uparrow \Rightarrow \text{dom}(\phi_{f(x)}) = \{0, \dots, i\}$ und damit endlich, also $f(x) \notin A$.

- $(\mathbb{N} \setminus K_0) \leq_m \{x \mid W_x \text{ ist endlich}\}$
- $\{x \mid W_x \text{ ist endlich}\} \not\leq_m K_0$
- $\{x \mid W_x \neq \emptyset\}$ ist RE-vollständig

9. (KW 50) Zeige $M \leq_m (\mathbb{N} \setminus M) \wedge M \in \text{RE} \Rightarrow M \in \text{REC}$.

Konstruiere M mit $M \leq_m (\mathbb{N} \setminus M)$ und $M \notin \text{REC}$

10 Turing-Maschinen

Motivation Turing-Maschine

- bisher: Rechnen mit Zahlen, jetzt: Rechnen mit Wörtern (Zeichenfolgen)

- stellt Verbindung her zw. Berechenbarkeit und Theorie der formalen Sprachen
- liefert ein genaueres Modell zur Messung des Ressourcenverbrauchs (in Komplexitätstheorie), (Rechnen mit beliebig großen Zahlen ist zu ungenau)
- es war nie beabsichtigt, Turing-Maschinen tatsächlich zu bauen (anders als bei goto-Programmen)
- aber die Natur macht es: (Umformungen von RNA)

TM: Semantik (im Grundsatz)

Grundsatz: ein Schritt einer Rechnung ist eine lokal beschränkte Speicher-Änderung.

- Speicher ist Zustand sowie Folge von Bändern
- Band ist Folge von Zellen
- jedes Band hat eine markierte Position (Kopfposition)
- ein Schritt besteht aus: Lesen, Schreiben, Bewegen
(evtl. über den Rand, dabei wird das Band verlängert)
- Zustandsmenge ist fixiert und endlich
- Zeichenvorrat (Zelleninhalt) ist fixiert und endlich
- Anzahl der Bänder ist fixiert und endlich
- jedes Band ist endlich, aber nicht beschränkt

TM: Syntax und Semantik (Rel. auf Konfig.)

Bezeichnungen für k -Band-Turing-Maschine:

- endliche Zustandsmenge Q , • endliches Alphabet Σ
- Leerzeichen $\square \notin \Sigma$, Bezeichnung $\Sigma_{\square} = \Sigma \cup \{\square\}$
- Band-Inhalte: $B(\Sigma) := \{f : \mathbb{Z} \rightarrow \Sigma_{\square} \text{ mit } f^{-1}(\Sigma) \text{ endlich} \}$
- endliche Zahl $k \geq 2$ (Anzahl der Bänder)

Die Konfigurationsmenge einer TM ist $Q \times (B(\Sigma) \times \mathbb{Z})^k$.

Das Programm einer TM besteht aus

- Initialzustand $q_i \in Q$, Finalzustand $q_f \in Q$
- Übergangstabelle $t : Q \times \Sigma_{\square}^k \hookrightarrow Q \times (\Sigma \times \{L, H, R\})^k$

Definiert Relation (partielle Funktion) \rightarrow_M auf Konfigurationen durch ...

TM: Beispiel

- mit 1 Arbeitsband.
- $\Sigma = \{0, 1\}$, Leerzeichen \square
- $Q = \{i, a, f\}$, initial: i , final: f
- für $x \in \Sigma$: $t(i, [x, p, y]) = (i, [(x, R), (x, R), (y, H)])$
 $t(i, [\square, p, y]) = (a, [(\square, H), (p, L), (y, H)])$
für $p \in \Sigma$: $t(a, [x, p, y]) = (a, [(x, H), (p, L), (p, R)])$
 $t(a, [x, \square, y]) = (f, [(x, H), (\square, H), (y, H)])$

berechnet Wortfunktion $\Sigma^* \rightarrow \Sigma^* : w \mapsto \text{reverse}(w)$

TM: Semantik (berechnete Wortfunktion)

für TM mit Einschränkung: jede Bewegung für Kopf 1 (Eingabe) und Kopf k (Ausgabe) ist $\in \{H, R\}$

- Eingabe ist read-only, Ausgabe ist write-only
- restliche Bänder heißen *Arbeitsbänder*

initiale Konfiguration für Eingabe $u \in \Sigma^*$:

- Zustand q_i , Band 1 enthält u (ab 0), sonst leer, Köpfe: 0

finale Konfiguration mit Ausgabe $v \in \Sigma^*$:

- Zustand q_f , Inhalt von Band k ist $\dots, \square, v, \square, \dots$

TM M berechnet $f_M : \Sigma^* \hookrightarrow \Sigma^*$ mit

$y = f_M(x)$, gdw. $\exists c : \text{initial}(x) \rightarrow_M^* c \wedge \text{final}(c) \wedge \text{output}(c) = y$

TM: Semantik (berechnete Zahlfunktion)

TM M mit Alphabet $\{1, \#, \dots\}$ (Zählzeichen, Trennzeichen)
berechnet Zahlfunktion $g : \mathbb{N}^n \hookrightarrow \mathbb{N}$ mit

- $g(x_1, \dots, x_n) = y$, gdw. $f_M(1^{x_1}\#1^{x_2}\#\dots\#1^{x_n}) = 1^y$

Wir nennen die so berechenbaren partiellen Funktionen Turing-berechenbar.

Genauer: $\text{Turing}_k :=$ die durch TM mit k Arbeitsbändern berechenbaren partiellen Funktionen, $\text{Turing} := \bigcup_{k \geq 0} \text{Turing}_k$

Ü: Nachfolger \in Turing, Addition \in Turing, Multipl. \in Turing

Nach der These von Church und Turing ist zu erwarten: Goto = While = Part = Turing,

Beweis durch Compiler von und nach Goto.

Turing \subseteq Goto

Satz: f ist TM-berechenbar $\Rightarrow f$ ist Goto-berechenbar.

- Beweis (Idee): simuliere Band mit Kopf durch $(l, r) \in \mathbb{N}^2$, d.h. zwei Register.
- Jede Zahl ist Wort zur Basis $b = 1 + |\Sigma|$.
- In l der Bandinhalt links vom Kopf, in r der Bandinhalt rechts vom Kopf (gespiegelt).
- Zeichen am Kopf lesen: r modulo b .
- Zeichen x schreiben und Kopf nach rechts: $l' := b \cdot l + x; r' = \lfloor r/b \rfloor$;
- für diese Rechnungen braucht man noch ein Register, insgesamt: k Bänder $\Rightarrow 2k + 1$ Register

Goto \subseteq Turing

Satz: f ist Goto-berechenbar $\Rightarrow f$ ist Turing-berechenbar.

Beweis (Idee):

- k Register $\rightarrow k$ Arbeits-Bänder
- Register i hat Inhalt $x \rightarrow$ Arbeits-Band i hat Inhalt 1^x
- Programmablaufsteuerung: Befehlsnummer (PC) im Zustand der TM merken

Ü: ergänze Details zur Herstellung der Initialkonfiguration, Ablesen des Resultates aus Finalkonfiguration

Simulation von Mehrbandmaschinen

Mehrere Arbeitsbänder sind nützlich, aber nicht nötig:

Satz: $\forall k \geq 1 : \text{Turing}_k = \text{Turing}_1$.

Beweis (Idee):

- Kodiere Inhalte der k Zellen auf Position p der Arbeitsbänder in ein einziges Zeichen aus Σ_{\square}^k .

Beachte bei Realisierung:

- es müssen dann immer alle Köpfe genau untereinander stehen
- es werden nicht die Köpfe, sondern die Bänder bewegt,
- eine simulierte Bewegung eines Kopfes ändert das gesamte simulierte Band

Maschinen mit wenigen Registern

- nach voriger Idee kann man Eingabe-, Ausgabe- und Arbeitsbänder in ein einziges Band kodieren
- diesen Bandinhalt in zwei Registern x, y verwalten
- zur Simulation eines Schrittes (Division, Multiplikation mit $|\Sigma| + 1$) benötigt man ein drittes Register z
- diese drei Register x, y, z kann man durch zwei simulieren: in a steht immer $2^x 3^y 5^z$, und b zum Rechnen.
- \Rightarrow das Halteproblem ist bereits für While- (oder Goto-) Programme mit 2 Registern (die nur Inc/Dec ausführen) unentscheidbar.
- ... für 1 Register entscheidbar (spezieller Kellerautomat)

Übung TM

1. Ein *einseitig unendliches* Band ist ein Band, dessen Kopf nur Positionen ≥ 0 einnimmt.

(bei unserer Def. der TM gilt: Ein- und Ausgabeband sind einseitig unendlich, Arbeitsbänder nicht.)

Begründen Sie, daß man ein unbeschränktes (d.h. zweiseitig unendliches) Band durch zwei einseitig unendliche Bänder schrittweise simulieren kann.

(„schrittweise“: ein Schritt der Original-Maschine \Rightarrow ein Schritt in der simulierenden Maschine)

2. Wir nennen einen TM-Befehl *stationär*, wenn dabei alle Köpfe stehenbleiben.

Beweisen Sie, daß jede TM-berechenbare Wortfunktion auch durch ein TM-Programm ohne stationäre Befehle berechnet werden kann.

11 Wortersetzungssysteme

Historische Motivation

- *Darstellungen* von Gruppen durch Generatoren und Relationen
- Bsp: $G = \langle a, b \mid a^2 = b^2 = (ab)^2 = 1 \rangle$
Symmetriegruppe des Rechtecks (Kleinsche 4-Gruppe)
(a, b sind Spiegelungen, was ist ab ?)
- Rechnen in G : $bab^3a^3ba =_G \dots =_G a^2ba$
Wortproblem: gegeben $G = \langle \dots \mid R \rangle$, u, v , gilt $u =_G v$?
- Felix Klein (1849–1925, in Leipzig: 1880–1886)
Erlanger Programm (der Geometrie), vgl. auch \equiv_m, \leq_m

SRS, Markov-Algorithmen (MA)

- (Syntax) Regelsystem $R \subseteq \Sigma^* \times \Sigma^*$
- (Semantik) Teilwort-Ersetzungs-Relation \rightarrow_R auf Σ^* $u \rightarrow_R v : \iff \exists p \in \Sigma^* : \exists (l, r) \in R : \exists q \in \Sigma^* : u = plq \wedge prq = v$
- Bsp. $\Sigma = \{1, A\}$, $R = \{(A1, 11A), (AE, \epsilon)\}$. Nf. von $k111$?
- Markov-Algorithmus := SRS R mit Strategie s
zur Auswahl genau eines Nachfolgers (z.B. leftmost)
- Andrej Markov Jr., 1903 – 1979

- die durch MA (R, s) berechnete Wortfunktion:
 $f_R(u) = v \iff \text{initial}(u) \rightarrow_{R,s}^* v \wedge \text{final}(v)$,
wobei $\text{initial}(w) := AwE$, $\text{final}(w) := w$ ist R -Normalform
- die durch Markov-Algorithmus berechnete Zahlfunktion:
kodiere Eingabe $x \in \mathbb{N}^k$ als $1^{x_1} \# \dots \# 1^{x_k}$

Vergleich MA — TM

- Satz: für jede Wortfunktion $f : \Sigma^* \leftrightarrow \Sigma^*$ gilt:
 f ist Markov-berechenbar $\iff f$ ist Turing-berechenbar
- Beweis-Plan \Rightarrow
 - w auf Arbeitsband
 - TM bestimmt Regel und Stelle der Anwendung
- Beweis-Plan \Leftarrow
 - kodiere alle Bänder in eines
 - übersetze Konfig (q, b, k) in Wort $b[\dots, k-1] q b[k, \dots]$
 - übersetze TM-Befehl $(q, x) \rightarrow (q', x', R)$
in SRS-Regel $qx \rightarrow x'q'$, entspr. für H, L (Übung)
 - für dieses R ist \rightarrow_R partielle Fkt. $\Sigma^* Q \Sigma^* \leftrightarrow \Sigma^* Q \Sigma^*$

Das Erreichbarkeitsproblem

$E_{\text{Markov}} := \{(R, u, v) \mid u \rightarrow_{R,\text{det}}^* v\}$, $E_{\text{SRS}} := \{(R, u, v) \mid u \rightarrow_R^* v\}$
Eigenschaften:

- $E_{\text{SRS}} \in \text{RE}$ (Beweis: Übung)
- $E_{\text{SRS}} \notin \text{REC}$

Beweis: zeige $K_{\text{Turing}} \leq_m E_{\text{Turing}} \leq_m E_{\text{Markov}} \leq_m E_{\text{SRS}}$

TM M hält (nach Def.) durch Erreichen des Zustandes q_f .

Konstruiere M' : wie M , aber für q_f Regeln zum Löschen aller Bänder, dann Halt in neuem, finalen Zustand q'_f

Dann $M(u) \downarrow \iff M'(u) = v_0 = \text{leeres Band (Bänder)}$

D.h. Reduktion (für \leq_m) durch $(M, u) \mapsto (M', u, v_0)$

12 Das Postsche Korrespondenzproblem

Motivation

- ein leicht zu definierendes kombinatorisches Problem
(eine Eigenschaft einer Folge von Paaren von Wörtern)
(„leicht“ heißt: Def. PCP ist kürzer als Def. TM)
- das trotzdem schwierig ist.
(„schwierig“ heißt: $\text{PCP} \notin \text{REC}$)
- ist Hilfsmittel für Beweise der Unentscheidbarkeit von Eigenschaften formale Sprachen, logischer Formeln, usw.
(„Hilfsmittel“ heißt: wir verwenden \leq_m)
- Emil Post (1897 – 1954)

Definition

- Eine *Instanz* des Postschen Korrespondenzproblems besteht aus:
endl. Menge Σ (Bsp. $\{0, 1\}$), endl. Menge Γ (Bsp. $\{a, b, c\}$),
zwei Morphismen (Wdhlg. Def.?) $f, g : \Gamma^* \rightarrow \Sigma^*$
Bsp. $f(a) = 10, f(b) = 101, f(c) = 1, g(a) = 1, g(b) = 010101, g(c) = 0$
- $w \in \Gamma^+$ heißt Lösung der Instanz, wenn $f(w) = g(w)$
- PCP ist die Menge der lösbaren PCP-Instanzen

Für das Beispiel gilt: ϵ ist keine Lösung. ab ist keine Lösung. Keine Lösung beginnt mit ab . Es gibt eine Lösung. Es gibt unendliche viele Lösungen.

$E_{\text{SRS}} \leq_m \text{PCP (Plan)}$

- Ziel: berechenbare Funktion f mit für alle R , Wörter u, v :
 $u \rightarrow_R^* v \iff f(R, u, v) \in \text{PCP}$
Sei $f(R, u, v) = I = (\Gamma, \Sigma, g, h)$

- Plan: w ist Lösung von I gdw.
Lösungswort $g(w) = h(w) = Au_0\#u_1\#\dots\#u_n\#E$
mit $u = u_0$ und $\forall i : u_i \rightarrow_R u_{i+1}$ und $u_n = v$
- $I = \frac{g}{h} \parallel \begin{array}{c|c|c|c|c|c|c|c|c|c} A & l_1 & \dots & l_{|R|} & v\#E & x_1 & \dots & x_k \\ \hline Au\# & r_1 & \dots & r_{|R|} & E & x_1 & \dots & x_k \end{array}$
mit $\Sigma = \{x_1, \dots, x_k\}$
- außer dem gewünschten Lösungswort
gibt es triviale Lösungen wegen der Kopier-Regeln.
- PCP \Rightarrow MPCP, Lösungen sollen mit ersten Paar beginnen

Das start-beschränkte PCP

(in der Literatur oft: „modifiziertes PCP“, MPCP)

zusätzlicher Parameter $a \in \Gamma$

MPCP = $\{(\Gamma, \Sigma, f, g, a) \mid \exists w : f(aw) = g(aw)\}$

Satz: MPCP \leq_m PCP

Beweis: $I : (\Gamma, \Sigma, f, g, a) \mapsto I' : (\Gamma \cup \{e\}, \Sigma \cup \{M, E\}, f', g')$

- neuer Buchstabe M soll an jeder zweiten Stelle von $f'(w') = g'(w')$ vorkommen,
- benutze Morphismen $L(x) = Mx, R(x) = xM$
- $\frac{f'}{g'} \parallel \begin{array}{c|c|c|c|c} L(f(a))M & R(f(b)) & \dots & E \\ \hline L(g(a)) & L(g(b)) & \dots & ME \end{array}$
- aw löst I gdw. awe löst I' , jede Lösung von I' beginnt mit a und endet mit e

PCP \notin REC

- wir haben nun gezeigt: $E_{SRS} \leq_m$ MPCP \leq_m PCP
- aus $E_{SRS} \notin$ REC folgt PCP \notin REC
- damit beweisen wir nun die Unentscheidbarkeit weiterer Probleme aus den Bereichen
 - formale Grammatiken
 - Logik

Eine unentscheidbare Eigenschaft von CFG

- Def. $\text{CFGine} = \{(G_1, G_2) \mid G_i \in \text{CFG}, L(G_1) \cap L(G_2) \neq \emptyset\}$
(context-free grammar intersection non-emptiness)
Satz: $\text{CFGine} \notin \text{REC}$. Beweis: $\text{PCP} \leq_m \text{CFGine}$
- Beweis: gegeben PCP-Instanz $I = (\Gamma, \Sigma, f, g)$, konstruiere
 - G_1 mit $L(G_1) = \{\text{reverse}(g(w))\#f(w) \mid w \in \Gamma^+\}$
 - G_2 mit $L(G_2) = \{\text{reverse}(u)\#u \mid u \in \Sigma^*\}$

Übung: untersuche Mitgliedschaft in REC, RE, coRE für

- $\{(G_1, G_2) \mid G_i \in \text{CFG}, L(G_1) \subseteq L(G_2)\}$
- $\{(G_1, G_2) \mid G_i \in \text{CFG}, L(G_1) = L(G_2)\}$
- $\{G \mid G \in \text{CFG}, L(G) = \Sigma^*\}, \{G \mid G \in \text{CFG}, L(G) = \emptyset\}$
- $\{G \mid G \in \text{CFG}, G \text{ ist eindeutig}\}$

13 Unentscheidbarkeit in der Logik

Prädikatenlogik

Das Allgemeingültigkeitsproblem APL ist

- Eingabe: PL-Formel F , ohne freie Variablen, in Sign. Σ ,
- Frage: F allgemeingültig? (gilt in jeder Σ -Struktur S)

Bsp: $(\exists x : \forall y : P(x, y)) \rightarrow (\forall y : \exists x : P(x, y))$

Satz: $\text{PCP} \leq_m \text{APL}$. Folg. $\text{APL} \notin \text{REC}$

Beweis-Plan: Aus PCP-Instanz mit Paaren $(ab, ba), \dots$ konstruiere Formel $(F_1 \wedge F_2) \rightarrow \exists R(x, x)$ mit

- $F_1 = R(a(b(e)), b(a(e))) \wedge \dots$
- $F_2 = \forall x, y : R(x, y) \rightarrow R(a(b(x)), b(a(x))) \wedge \dots$

Arithmetik

- Die *Theorie* einer Struktur S ist die Menge der Formeln F mit $S \models F$.

- Die *Theorie der Arithmetik* \mathbf{TA} besteht aus allen wahren Aussagen über \mathbb{N} in der Signatur $\{0/0, 1/0, +/2, \cdot/2\}$
Bsp. $(\exists x : \forall y : x \neq 2 \cdot y) \in \mathbf{TA}$
- Satz (1931, Kurt Gödel, 1906–1978): $\mathbf{TA} \notin \text{REC}$.
Beweis: $E_{\text{While}} \leq_m \mathbf{TA}$, denn
die Zustandsübergangsrelation \rightarrow_P eines While-Programms kann *arithmetisiert* werden.
Für $P = \text{while}(x)Q$ verwende
 $\exists t, c : c$ ist Kodierung einer Folge von t Zuständen ...

Unvollständigkeit von Beweissystemen

Bezeichnung: eine PL-Formel F heißt

- *wahr*, wenn $\forall S : S \models F$ (gilt in jeder Struktur)
- *ableitbar* (durch Inferenzsystem (Axiomen und Regeln))

Satz: Menge der ableitbaren Formeln $\in \text{RE}$

Beweis: Inf.-Syst. ist endlich und jede Ableitung ist endlich.

Bezeichnung: ein Inferenzsystem heißt

- *korrekt*, wenn jede ableitbare Formel wahr ist
- *vollständig*, wenn jede wahre Formel ableitbar ist

Satz (Gödel): jedes korrekte Inferenzsystem für die Arithmetik ist unvollständig.

Bew.: $\text{co-TA} \leq_m \mathbf{TA}$ (wg. $F \mapsto \neg F$), $\mathbf{TA} \in \text{RE} \cap \text{coRE} = \text{REC}$

Übungen KW51

1. In der Kleinschen 4-Gruppe gilt: $ab = ba$. Beweise durch
 - (a) geometrischen Anschauung,
 - (b) eine Gleichungskette unter Benutzung der Relationen
2. Für das SRS $\{ab \rightarrow baa\}$: Normalform von $a^k b$, von ab^k .
3. Ergänze Simulation der TM durch MA (Kopfbewegungen H, L)
4. autotool-Aufgabe PCP

5. Für PCP mit $|\Sigma| = 1$:

- Geben Sie je eine lösbare und eine nicht lösbare Instanz mit $|\Gamma| = 3$ an.
- Geben Sie ein Entscheidungsverfahren an.

6. Die folgende Variante des PCP ist entscheidbar:

- Eingabe: Morphismen $f, g : \Gamma^* \rightarrow \Sigma^*$,
- Lösung: Wörter $w_1, w_2 \in \Gamma^+$ mit $f(w_1) = g(w_2)$

(Hinweis: endliche Automaten)

7. Aufgaben auf Folie „eine unentsch. Eig. von CFG“

Hinweis: $\Sigma^* \setminus L(G_2)$ ist auch kontextfrei.

(1. warum? 2. was nützt es?)

14 Orakel und Nichtdeterminismus

Motivation

- bisher: deterministische Rechnungen
 - Rechnung ist Pfad
 - ist erfolgreich, falls finaler Zustand erreicht wird
- jetzt: Such-Aufgaben (nichtdeterministische Rechnungen)
 - Rechnung ist Baum (d.h., mglw. mehrfach verzweigt)
 - ist erfolgreich, falls ein finales Blatt existiert
- alternative Sichtweise: Orakel-Berechnung
 - ein Orakel beschreibt den Weg zum finalen Blatt,
 - deterministische Maschine verifiziert diesen Weg

Eine Charakterisierung von RE

- $M \in \text{RE} \iff \exists i : M = \text{dom } \phi_i$
- $x \in M \iff \exists y : \text{Rechn. } \phi_i(x) \text{ hält nach genau } y \text{ Schritten}$
- $\{C(x, y) \mid \text{Rechn. } \phi_i(x) \text{ hält nach genau } y \text{ Schr.}\} \in \text{REC}$
- Satz: $\forall M \subseteq \mathbb{N} : M \in \text{RE} \iff \exists M' \in \text{REC} : \forall x : x \in M \iff \exists y : C(x, y) \in M'$
- Ü: Beweis für \Leftarrow
- Bezeichnungen: y ist *Orakel-Wort* oder *Zertifikat* für x .
- Ü: falls $\exists M' \in \text{REC} : \forall x : x \in M \iff \exists y : C(x, y) \in M' \wedge y \leq x$, dann $M \in \text{REC}$

Orakel-Maschinen

- Def: eine *Orakel-Maschine* A hat Eingabeband (Alphabet Σ), Orakelband (Alphabet Γ) und Arbeitsbänder
- Def: die von A akzeptierte Sprache $\text{Lang}_{\text{Acc}}(A) \subseteq \Sigma^*$:
Menge aller Eingabewörter x , für die ein Orakelwort y existiert, so daß Rechnung $M(x, y)$ erfolgreich hält.
- nach dieser Def: das Orakel sieht x und schreibt y , dann rechnet (*verifiziert*) A (deterministisch)
- alternative Sichtweise: y ist unbekannt, bei jedem Lesen eines unbekanntes Zeichens von y verzweigt die Rechnung: A ist *nicht-deterministische* TM
- die Rechnung $A(x)$ ist ein Baum T , Eingabe wird akzeptiert, wenn T ein erfolgreiches Blatt enthält

15 Komplexitätstheorie

Motivation

- bis jetzt: Berechenbarkeit als *qualitativer* Begriff
(eine Funktion ist berechenbar oder nicht, eine Menge ist entscheidbar oder nicht)
- ab jetzt: *quantitative* Untersuchung:
für berechenbare Funktionen/entscheidbare Mengen:
mit welchem Aufwand läßt sich Rechnung durchführen?
- Komplexität sowohl für det. als auch für nichtdet. Berechnungsmodelle (Suchprobleme)
- verwendet Methoden der Berechenbarkeitstheorie (insb. Reduktion, Vollständigkeit)

Definition

- Ressourcen:
z.B. Rechenzeit, Speicherplatz, Kommunikationsaufwand
- Komplexität eines Algorithmus (eines Programmes, einer Maschine):
Ressourcenverbrauch als *Funktion* der Eingabegröße
Bsp: die Komplexität von Bubblesort ist quadratisch.
- Komplexität eines Problems: Komplexität für „besten“ Algorithmus, der das Problem löst.
Bsp: die Komplexität des Sortierens ist $\in \Theta(n \mapsto n \cdot \log n)$

Komplexitätstheorie ist die Lehre von der Schwierigkeit von *Problemen*.

Eigenschaften von Zahlen

- Zahlen sind hier: ganz, nichtnegativ und binärkodiert
- zusammengesetzte Zahlen
 $\text{COMP} = \{x \mid \exists y, z \geq 2 : x = y \cdot z\}$
- Primzahlen
 $\text{PRIMES} = \{x \mid x \geq 2\} \setminus \text{COMP}$

- Quadratzahlen
 $\text{SQUARES} = \{x \mid \exists y : x = y^2\}$
- Motivation: u.a. Kryptographie
 RSA-Schlüssel $\in \text{COMP}$, aber die Ursache (die Faktoren) soll geheim bleiben

Das Erfüllbarkeitsproblem

- AL = die Menge der aussagenlogischen Formeln
- $b \models F$: die Belegung b erfüllt die Formel F
- $\text{SAT} = \{F \mid F \in \text{AL}, \exists b : b \models F\}$
- Bsp. $((x \rightarrow \neg y) \wedge (x \leftrightarrow y)) \in \text{SAT}, (x \wedge \neg x) \notin \text{SAT}$
- Spezialfälle (durch syntaktische Einschränkungen)
 - CNF-SAT:
wie oben und F ist in konjunktiver Normalform
 - k -CNF-SAT:
... mit $\leq k$ Literalen je Klausel
- Motivation: Entwurf und Überprüfung von digitalen Schaltungen

Boolesche Schaltkreise

- Schaltkreis ist DAG mit:
 - Startknoten (keine Vorgänger): Eingaben
 - andere Knoten: markiert mit Boolescher Op. (\wedge, \vee, \neg)
 - Endknoten (keine Nachfolger): Ausgaben
- jeder Knoten realisiert Bool. Fkt. der Eingabe
- Schaltkreis-Erfüllbarkeit $\text{CIRCSAT} = \{C \mid C \text{ ist unärer Schaltkreis (mit einer Ausgabe)} \wedge \exists \text{ Eingabe } e \text{ mit Ausgabe } C(e) = 1, \text{ Schreibweise } e \models C\}$
- Satz: $\text{CIRCSAT} \in \text{NP}$,
 Beweis: das Orakel rät die Eingabe e , dann kann $C(e)$ in Polynomialzeit berechnet werden (z.B. schichtweise)

Das Färbungsproblem

- eine k -Färbung eines Graphen $G = (V, E)$ ist eine Abbildung $f : V \rightarrow \{1, 2, \dots, k\}$
- die Färbung f von $G = (V, E)$ ist konfliktfrei, wenn $\forall xy \in E : f(x) \neq f(y)$
- $k\text{COL} := \{G \mid \exists f : f \text{ ist konfliktfreie } k\text{-Färbung von } G\}$
- Bsp: $K_3 \notin 2\text{COL}$, $C_5 \notin 2\text{COL}$,
Ü: finde G mit: G enthält keinen K_3 und $G \notin 3\text{COL}$
Ü: jeder Knoten von G hat $< k$ Nachbarn $\Rightarrow G \in k\text{COL}$.
- Motivation: Ressourcenzuordnungsprobleme,
z.B. Frequenzbereiche zu Funkzellen

Bemerkung zur Genauigkeit

bisher:

- wg. These von Church und Turing war das Berechnungsmodell beliebig
- wg. Gödelisierung war die Art der Eingabe (Zahlen, Wörter usw.) beliebig

jetzt:

- wg. exakter Ressourcenmessung muß ein Modell fixiert werden (Turingmaschine)
- wg. Komplexität als Fkt. der Eingabegröße
muß „Größe“ exakt definiert werden
(Länge des Wortes auf dem Eingabeband, Länge der Binärcodierung einer Zahl)

Deterministische Zeit, Bps.: P

- für $f : \mathbb{N} \rightarrow \mathbb{N}$ bezeichnet $\text{DTIME}_{\text{Alg}}(f)$
die Menge der TM M mit $\forall x \in \text{Lang}_{\text{Acc}}(M) \iff M$ akzeptiert x nach $\leq f(|x|)$
Schritten.
- für $f : \mathbb{N} \rightarrow \mathbb{N}$ bezeichnet $\text{DTIME}_{\text{Prob}}(f)$
die Menge der Sprachen L mit $\exists M \in \text{DTIME}_{\text{Alg}}(f) : L = \text{Lang}_{\text{Acc}}(M)$.
- Abkürzung $\text{DTIME} = \text{DTIME}_{\text{Prob}}$

Satz: f berechenbar \Rightarrow $\text{DTIME}(f) \subseteq \text{REC}$

- für Menge F von Funktionen: $\text{DTIME}(F) = \bigcup_{f \in F} \text{DTIME}(f)$
- Abkürzung: $\text{P} = \text{PTIME} = \text{DTIME}(\text{Menge der Polynome})$
- $\{w \mid w \in \{0, 1\}^*, w = \bar{w}\} \in \text{P}$, $\text{CFL} \subseteq \text{P}$, $2\text{COL} \in \text{P}$, $2\text{SAT} \in \text{P}$

Nichtdeterministische Zeit, Bsp: NP

- für $f : \mathbb{N} \rightarrow \mathbb{N}$ bezeichnet $\text{NTIME}_{\text{Alg}}(f)$ die Menge der Orakel-TM M mit $\forall x : x \in \text{Lang}_{\text{Acc}}(M) \iff$ es gibt ein Orakelwort y mit: $M(x, y)$ akzeptiert in $\leq f(|x|)$ Schritten.
(dabei werden $\leq f(|x|)$ Zeichen von y gelesen)
- entsprechend $\text{NTIME}_{\text{Prob}}(f)$, $\text{NTIME}(f)$, $\text{NTIME}(F)$
- $\dot{\cup}$: f berechenbar $\Rightarrow \text{NTIME}(f) \subseteq \text{REC}$
- Abkürzung $\text{NP} = \text{NPTIME} = \text{NTIME}(\text{Polynome})$
- Bsp. $\text{COMP} \in \text{NP}$, $\text{SAT} \in \text{NP}$, $\forall k : k\text{COL} \in \text{NP}$
- Satz: $A \in \text{NP} \wedge B \in \text{NP} \Rightarrow (A \cup B) \in \text{NP}$
- Satz: $\text{P} \subseteq \text{NP}$ (trivial), Frage $\text{P} = \text{NP}$ (schwierig: 10^6 \$, <http://www.claymath.org/millennium-problems/p-vs-np-problem>)

Die Klasse coNP

- Def: $\text{coNP} = \{L \mid (\Sigma^* \setminus L) \in \text{NP}\}$
- Bsp: $\text{PRIMES} \in \text{coNP}$ wegen $\text{COMP} \in \text{NP}$
- es gilt auch $\text{PRIMES} \in \text{NP}$, Beweis benötigt etwas Zahlentheorie
- $\text{PRIMES} \in \text{P}$,
Manindra Agrawal, Neeraj Kayal, Nitin Saxena, 2004.
<http://annals.math.princeton.edu/2004/160-2/p12>

Vergleich: Berechenbarkeit/Komplexität

bis jetzt:

- es gibt Analogien zwischen:
 - Berechenbarkeit: REC, RE, coRE
 - Komplexität: P, NP, coNP
- der Unterschied ist, daß man $REC \neq RE$, $RE \neq coRE$ und $RE \cap coRE = REC$ beweisen kann,
die entsprechenden Komplexitätsaussagen bisher nicht

nächste VL:

- Reduktion (Vergleich der Komplexität von Problemen)
- Vollständigkeit (zur Def. der schwersten Probleme einer Klasse)

Übungsaufgaben P/NP

Markierte Aufgaben besonders empfohlen zur Diskussion in Übung KW 54.

1. (KW54) Beweise: für alle $M' \in REC$ und totale berechenbare $f : \mathbb{N} \rightarrow \mathbb{N}$ gilt $\{x \mid \exists y : C(x, y) \in M' \wedge y \leq f(x)\} \in REC$

auf Deutsch: wenn die Größe der Orakelzahlen durch eine berechenbare Funktion beschränkt ist, dann ist jede mit diesem Orakel entscheidbare Menge auch ohne Orakel entscheidbar.

2. FP und LOOP:

(a) Geben Sie eine Funktion $f \in LOOP$ an, die nicht in Polynomialzeit berechenbar ist (Hinweis: z.B., weil sie zu schnell wächst)

(b) $LOOP^-$ -Programme: wie LOOP, aber

- zusätzlich ein Befehl $Copy(x, y)$ mit Semantik $y := x$
- Falls $Inc(x)$ in einer Schleife vorkommt, dann heißt x vergiftet. Falls $Copy(x, y)$ und x vergiftet, dann auch y vergiftet. Vergiftetes Register darf nicht als Schleifenzähler benutzt werden.

Beweise: Addition, Multiplikation $\in LOOP^-$.

Beweise: $LOOP^-$ -Programme haben polynomielle Laufzeit

3. (KW 54) Abschlußeigenschaften

- (a) ist P abgeschlossen unter: Vereinigung, Durchschnitt, Komplement? (trivial)
- (b) ... Verkettung? (ja.) Stern?
- (c) die gleichen Fragen für NP .

4. Zahlen

- (a) $SQUARES \in NP$ (trivial), $SQUARES \in P$
- (b) $PRIMES \in NP$ (Satz von Fermat, primitive Wurzeln)

5. Graphen

- (a) 3COL: autotool
- (b) gesucht: ein G mit $G \notin 3COL$ und G enthält kein K_3 (Dreieck)
- (c) (KW 54) für alle G mit $\deg(G) < k$ gilt $G \in kCOL$.
Hinweis: geben Sie einen Algorithmus an, der unter dieser Voraussetzung eine k -Färbung für G konstruiert.
- (d) Für welche Graphen gilt: G hat Maximalgrad k und $G \notin kCOL$?

6. Logik

- (a) CNF-SAT: autotool
- (b) (KW 54) 2CNFSAT $\in P$ (schreibe jede Klausel als Implikation und betrachte einen dazu passenden Graphen)
- (c) DNFSAT $\in P$ (disjunktive Normalform)

16 NP-Vollständigkeit

Motivation

nach bisherigen Definitionen/Beispielen:

- P : Klasse der (auf sequentiellen Maschinen) effizient lösbaren Probleme
- NP : enthält häufig vorkommende Suchprobleme

für (neues) Problem L wüßte man gern: $L \in P$ oder $L \notin P$?

- bis heute ist kein $L \in NP \setminus P$ bekannt, obwohl es viele Kandidaten gibt.

- Bekannt ist jedoch eine Charakterisierung der *schwersten* Probleme in NP:
die Klasse NPC der *NP-vollständigen Probleme*

Polynomialzeit-Reduktion \leq_P

- Def: $\text{FTIME}(f) :=$ die in Zeit $\leq f(|x|)$ auf DTM berechenbaren Fkt., $\text{FP} := \text{FTIME}(\text{poly})$.
- Def: $A \leq_P B := \exists f \in \text{FP} : \forall x \in \Sigma^* : x \in A \iff f(x) \in B$.
- Bsp: $\text{DHC} \leq_P \text{HC}$.
 $\text{HC} := \{G \mid G \text{ ist ungerichteter Graph und } G \text{ enthält Kreis durch alle Knoten}\}$
(Hamiltonian Circuit)
 $\text{DHC} := \{G \mid G \text{ ist gerichteter Graph und } G \text{ enthält gerichteten Kreis durch alle Knoten}\}$ (Directed HC)
Beweis: $f(V, E) = (V \times \{1, 2, 3\}, E')$ mit $E' = \{(v, 1)(v, 2) \mid v \in V\} \cup \{(v, 2)(v, 3) \mid v \in V\} \cup \{(u, 3)(v, 1) \mid (u, v) \in E\}$.
zu zeigen sind für f : Korrektheit, Laufzeit

\leq_P ist transitiv

- vgl.: \leq_m ist transitiv,
- Beweis: sei $A \leq_P B$ mit Reduktionsfunktion f , $B \leq_P C$ mit Reduktionsfunktion g .
zeige: $A \leq_P C$ durch Reduktionsfunktion $h : x \mapsto g(f(x))$
Korrektheit ist offensichtlich, Laufzeit: folgt aus:
- Satz: $f \in \text{FP} \wedge g \in \text{FP} \Rightarrow (x \mapsto g(f(x))) \in \text{FP}$.
- Bew.: sei $f \in \text{FTIME}(n \mapsto c \cdot n^k), g \in \text{FTIME}(n \mapsto d \cdot n^l)$.
 - die Ausgabe von f ist die Eingabe von g
 - Ausgabenlänge \leq Rechenzeit

daraus folgt $(x \mapsto g(f(x))) \in \text{FTIME}(n \mapsto \dots \cdot n^{\dots})$

Abschluß-Eigenschaften von \leq_P

- Satz: $A \leq_P B \wedge B \in P \Rightarrow A \in P$
- Beweis: benutze Abschluß von FP bzg. Komposition für
 - die Reduktionsfunktion f von A nach B
 - die charakteristische Fkt. c_B von B
- Satz: $A \leq_P B \wedge B \in NP \Rightarrow A \in NP$
- Beweis: sei f die Reduktionsfunktion von A nach B ,
 c_B wird orakel-berechnet $M : (x, y) \mapsto \{0, 1\}$.
Dann wird c_A orakel-berechnet durch $(x, y) \mapsto M(f(x), y)$
zu betrachten sind: Korrektheit, Laufzeit, Orakelgröße.

vgl. $A \leq_m B \wedge B \in REC \Rightarrow A \in REC$, $A \leq_m B \wedge B \in RE \Rightarrow A \in RE$.

NPc und Satz von Cook

- Def: B heißt NP-schwer, gdw. $\forall A \in NP : A \leq_P B$.
- Def: B NP-vollständig, gdw. B NP-schwer und $B \in NP$
- NPc := $\{B \mid B \text{ ist NP-vollständig}\}$
- Satz (Steven Cook, 1971): SAT \in NPc
- wir zeigen zunächst: CIRCSAT \in NPc
- Bew: CIRCSAT \in NP wurde schon gezeigt.
Bleibt zu zeigen: CIRCSAT ist NP-schwer. Dazu:
gegeben $A \in NP$, zu zeigen ist $A \leq_P$ CIRCSAT.

CIRCSAT \in NPc

- Bew: gegeben $A \in \text{NP}$, zu zeigen ist $A \leq_P \text{CIRCSAT}$.
- $A = \text{Lang}_{\text{Acc}}(M)$ für O.-TM M mit Zeitschranke $s \in \text{poly}$
- Reduktionsfunktion $f : x \mapsto \text{Schaltkreis } C$, der Konfigurationsfolge einer erfolgreichen Rechnung von M mit $\leq s(|x|)$ Schritten bei Eingabe x und Orakelwort y beschreibt, d.h., $x \in A \iff \exists y : C(x, y) = 1$
- Eingangsknoten: Kodierung von x, y
andere Knoten: indiziert durch Platz p und Zeit t , $K(p, t)$ enthält Kodierung von Zeichen, Kopfposition, Zustand
Ausgabeknoten: akzeptierender Zustand wurde erreicht
- C kann in FP konstruiert werden (Zeit und Platz sind $\leq s(|x|)$)

CIRCSAT \leq_P CNFSAT (Tseitin-Transformation)

Gregory Tseitin, 1966

- *Plan*: aus Schaltkreis C wird Formel G konstruiert
mit Knoten von $C = \text{Var}(G)$
und $\forall e : e \models C \iff \exists b : e \subseteq b \wedge b \models G$.
(jedes Modell von G ist Erweiterung eines Modells von C und jedes Modell von C läßt sich zu Modell von G erweitern)
- *Realisierung*: für jeden nicht-Eingabeknoten k von C mit Vorgängern k_1, k_2, \dots und Verknüpfung f :
Menge von CNF-Klauseln, die äquivalent ist zu $k \leftrightarrow f(k_1, k_2, \dots)$.

Tseitin-Transformation. Einzelheiten

- Einzelheiten, Bsp $f = \wedge$.
 $\bar{k} \vee k_1, \bar{k} \vee k_2, \dots, k \vee \bar{k}_1 \vee \bar{k}_2 \vee \dots$
(andere Verknüpfungen: Übung)
- Folgerung: CIRCSAT \leq_P CNFSAT \leq_P SAT
- CNFSAT ist NP-schwer, SAT ist NP-schwer
- Übung: SAT \leq_P 3CNFSAT. Hinweis: SAT \leq CIRCSAT und dann den Schaltkreis umformen, so daß bei unsere Reduktion CIRCSAT \leq_P CNFSAT nur 3-Klauseln entstehen

Cook und Tseitin in der Praxis

wenn sich ein Anwendungsproblem L als NP-vollständig herausstellt, dann folgt:

- es gibt derzeit keinen effizienten Algorithmus für L
- $L \leq_P \text{SAT} \leq_P \text{CNFSAT}$, d.h. man kann L lösen, wenn man CNFSAT lösen kann

... und gute CNFSAT-Solver gibt es tatsächlich

- Niklas Een, Niklas Sörensson: <http://minisat.se/>
- Wettbewerbe: <http://satcompetition.org/>

Einzelheiten: siehe VL Constraint-Programmierung <http://www.informatik.uni-leipzig.de/~waldmann/>

Übungsaufgaben NP, SAT

- Es gilt: zu jeder Formel F gibt es eine äquivalente Formel G in disjunktiver Normalform. Es gilt auch: $\text{DNFSAT} \in \text{P}$. Warum folgt daraus nicht $\text{SAT} \in \text{P}$?
Bem: $\text{SAT} \in \text{P}$ ist damit nicht ausgeschlossen, es geht nur darum, daß es so jedenfalls nicht folgt.
- (KW 55) Für den Schaltkreis für die Formel $(x_1 \wedge x_2) \vee x_3$:
Führe die Tseitin-Transformation durch.
- gib eine CNF an für „genau eine von (x_1, \dots, x_n) ist wahr“.
(z.B. $n = 4, n = 8$)
- gib, falls möglich, eine kleinere erfüllbarkeitsäquivalente CNF dafür an (d.h.: mit Zusatzvariablen, und so, daß die Tseitin-Spezifikation wahr ist)
- (KW 55) Def. $A \equiv_P B$ gdw. $A \leq_P B$ und $B \leq_P A$.
Seien $A, B \in \text{P}$. Wann gilt und gilt nicht $A \equiv_P B$?
- (KW 55) zeige $A \in \text{NPc} \wedge B \in \text{NPc} \Rightarrow A \equiv_P B$
- (KW 55) Für $A, B \subseteq \Sigma^*$ gilt $A \leq_P B \iff (\Sigma^* \setminus A) \leq_P (\Sigma^* \setminus B)$.

- man könnte definieren: B ist P-vollständig bzgl \leq_P gdw. $B \in P$ und $\forall A \in P : A \leq_P B$.
Das ist aber nicht interessant, denn sehr viele Mengen aus P haben diese Eigenschaft — welche nicht?
- (KW 55) Es gilt $3COL \leq_P SAT$ (warum? erfordert keine Rechnung, sondern nur die Kombination von zwei bereits bekannten Aussagen). Gib explizit eine Reduktionsfunktion an, die in P-Zeit aus einem Graphen G eine Formel F berechnet mit $F \in 3COL \iff G \in SAT$. Hinweis: für jeden Knoten 3 Variablen.

17 Weitere NP-vollständige Probleme

Motivation, Vorgehen

Motivation: $L \in NPC$ bedeutet $L \in NP \wedge \forall L' \in NP : L' \leq_P L$

- es gibt derzeit keinen effizienten Algorithmus für L (sonst auch für SAT, und das wäre eine Sensation)
- es hat auch wenig Sinn, einen solchen zu suchen
- besser: Aufgabenstellung einschränken o. variieren

Informatiker muß deswegen $L \in NPC$ schnell erkennen, um sinnlose Arbeit zu vermeiden.

- $L \in NP$ ist meist offensichtlich,
- statt $\forall L' \in NP : \dots$ reicht *ein* $L' \in NPC$ (warum?)
- deswegen braucht man Beispiel-Probleme aus NPC

Vertex Cover

- Def: Menge M heißt *Knoten-Überdeckung* von $G = (V, E) \iff M \subseteq V \wedge \forall e \in E : \exists v \in M : v \in e$
(M ist Menge von Knoten, die jede Kante überdeckt)
- Def: $VC = \{(G, k) \mid \exists M : M \text{ ist Knotenüberdeckung von } G \wedge |M| \leq k\}$
- Satz: $VC \in NPC$

- Beweis: $VC \in NP$ ist klar (Orakel liefert M)
zeigen $3SAT \leq_P VC$
(K_2 für jede Variable, K_3 für jede Klausel)
- Ü: ist die „3“ hier wirklich nötig?

Edge Cover

- Def: Menge M heißt *Kanten-Überdeckung* von $G = (V, E) \iff M \subseteq E \wedge \forall v \in V : \exists e \in M : v \in e$
(M ist Menge von Kanten, die jeden Knoten überdeckt)
- Def: $EC = \{(G, k) \mid \exists M : M \text{ ist Kantenüberdeckung von } G \wedge |M| \leq k\}$
- Satz: $EC \in P$
Beweis: evtl. Übung

3-Färbbarkeit

- Satz: $3COL \in NPc$
- Beweis: $3SAT \leq_P 3COL$
 - eine $K_3 \{0, \text{Falsch}, \text{Wahr}\}$
 - für jede Variable v ein $K_3 : \{v, \neg v, 0\}$
 - für jede Klausel $l_1 \vee l_2 \vee l_3$: 6 Knoten $K_3 - K_3$,
4 Endpunkte verbunden mit $l_1, l_2, l_3, \text{Falsch}$

Lemma: in jeder konfliktfreien 3-Färbung: die Nachbarn der vier Endpunkte des $K_3 - K_3$ haben nicht die gleiche Farbe.

- vgl. mit Ü: $3COL \leq_P SAT$
- Ü: Def: $COL = \{(G, k) \mid G \in kCOL\}$, Satz: $COL \in NPc$

Rucksack (Subset Sum)

Definition:

- Instanz: Zahlen $a_1, \dots, a_n, b \in \mathbb{N}$
- Lösung: Teilmenge $I \subseteq \{1, \dots, n\}$ mit $b = \sum_{i \in I} x_i$

Satz: Knapsack \in NPc, Beweis: 3SAT \leq_P Knapsack

Konstruktion: F mit n Variablen, m Klauseln,

- für $x \in \{a_1, \dots, b\}$: decimal(x) $\in \{0, 1, \dots, 4\}^*$,
- decimal(b) = $4^m 1^n$
- Gewichte a_i für: positive Vorkommen von Variable in Klausel, negative Vorkommen, Zusatzgewichte
- Überträge kommen nach Konstruktion nicht vor

Hamiltonkreis

- Satz: HC \in NPc
- Beweis: VC \leq_P HC

Handlungsreisender (TSP)

(travelling salesperson)

Formulierung als eingeschränktes Optimierungsproblem:

- Instanz: Matrix $D \in \mathbb{N}^{n \times n}$, Schranke $K \in \mathbb{N}$
- Lösung: Permutation p von $[1, \dots, n]$ (Rundreise)
- Maß: $c(D, p) = \sum_{i=1}^n D(p(i), p((i \bmod n) + 1))$

Formulierung als Sprache (Entscheidungsproblem):

TSP = $\{(D, K) \mid \exists \text{Permutation } p : c(p, D) \leq K\}$

Satz: TSP \in NPc

Beweis: HC \leq_P TSP

Ü: weitere Optimierungsprobleme <http://www.nada.kth.se/~viggo/problemlist/compendium.html>

Starfree Regexp Inequivalence

- Def: R_0 := reguläre Ausdrücke mit: Buchstabe, Verkettung, Vereinigung (kein Stern)
 - Def: $SRI = \{(X, Y) \mid X, Y \in R_0 \wedge \text{Lang}(X) \neq \text{Lang}(Y)\}$
 - Ü: $SRI \in \text{NP}$
- Clique, Subgraph Isomorphism, Graph Isomorphism

Übungsaufgaben KW56

- $\text{VC} \in \text{NPc}$
- zeige: $\text{planar 3COL} \in \text{NPc}$ mittels $3\text{COL} \leq_P \text{planar3COL}$
Dazu jede Kreuzung von zwei Kanten durch einen geeigneten Teilgraphen ersetzen.
- Dominating Set:
 $M \subseteq V$ ist dominierende Menge in $G = (V, E)$, falls $\forall v \in V \setminus M : \exists u \in M : vu \in E$
 $\text{DS} := \{(G, k) \mid \exists M : |M| \leq k \wedge M \text{ ist dominierend in } G\}$
 - zeige durch Beispiel, daß $\text{DS} \neq \text{VC}$ und $\text{DS} \neq \text{EC}$
 - Zeige $\text{DS} \in \text{NPc}$ durch geeignete Reduktion
- Edge Cover (EC) $\in \text{P}$

18 Zusammenfassung, Ausblick

Themen

- Berechnungsmodelle
Goto, While, Loop, Markov, Turing
konkrete und abstrakte Sytax, Semantik (Interpreter), Äquivalenzen (Compiler)
- Berechenbarkeit
 REC , $\phi_x(y)$, K , K_0 , Rice, PCP, RE, \leq_m , Vollständigkeit
- Komplexität
Nichtdeterminismus, Zeit, Platz, P, NP, \leq_P , NPc
SAT, COL, VC

Beweisverfahren: Diagonalisierung

Verfahren:

- Aufzählung einer Menge von Funktionen $F = \{f_0, f_1, \dots\}$
- modifizierte Diagonalfunktion $d' : x \mapsto \text{change}(f_x(x))$,
- $d' \in F \Rightarrow \exists i : d' = f_i \Rightarrow d'(i) = \text{change}(f_i(i)) \stackrel{?}{=} f_i(i) = d'(i)$

Übung/Wiederholung: wende an für $F = \dots$

- alle Funktionen $\mathbb{N} \rightarrow \mathbb{N}$
- alle Polynome
- alle totale berechenbaren Funktionen $\mathbb{N} \rightarrow \mathbb{N}$
- alle partiellen berechenbaren Funktionen $\mathbb{N} \leftrightarrow \mathbb{N}$
- alle primitiv rekursiven Funktionen
- alle in $\text{DTIME}(f)$ berechenbaren Funktionen

Algorithmen-Entwurfsmuster: Dovetailing

Verfahren:

- zähle \mathbb{N}^2 auf in Reihenfolge monoton steigender $C(x, y)$.

(Wdhlg.) Anwendungen:

- $M \in \text{RE} \iff \exists x : M = \text{dom}(\phi_x)$
- $\{x \mid W_x \text{ ist unendlich}\} \leq_m \{x \mid W_x = \mathbb{N}\}$

Weitere Aspekte der Komplexitätstheorie

- Platz-Schranken (oberhalb von \mathbf{P} , schwere Suchprobleme)
- Schaltkreise (unterhalb von \mathbf{P} , parallele Algorithmen)
- probabilistische Maschinen BPP (mit beschränkter Fehlerhäufigkeit) (kann Prob- \mathbf{P} mehr als \mathbf{P} ?)
- Quanten-Computer (BQP) (Beziehung zu NP?)

- Kryptografie (z.B. zero-knowledge proofs)
- *implizite* Komplexität von Programmen (syntaktische oder andere statische Bedingungen (Typen) P)
- Dexter Kozen: Theory of Computation, Springer 2006
- S. Arora, B.Barak: Computational Complexity, CUP 2009
- Scott Aaronson: <https://www.scottaaronson.com/blog/>

19 Die Ressource Speicherplatz

Deterministischer Platz

- wir messen den Platz *nur auf dem Arbeitsband* (nicht Eingabe-, nicht Ausgabeband)
- für $f : \mathbb{N} \rightarrow \mathbb{N}$ bezeichnet $\text{DSPACE}_{\text{Alg}}(f)$ die Menge der DTM M mit $\forall x \in \text{Lang}_{\text{Acc}(M)} \iff M$ akzeptiert x mit Arbeitsband der Breite $\leq f(|x|)$
- für $f : \mathbb{N} \rightarrow \mathbb{N}$ bezeichnet $\text{DSPACE}_{\text{Prob}}(f)$ die Menge der Sprachen L mit $\exists M \in \text{DSPACE}_{\text{Alg}}(f) : L = \text{Lang}_{\text{Acc}(M)}$
- Abkürzung $\text{DSPACE} = \text{DSPACE}_{\text{Prob}}$
- für Menge F von Funkt.: $\text{DSPACE}(F) = \bigcup_{f \in F} \text{DSPACE}(f)$
- Abkürzung $\text{PSPACE} = \text{DSPACE}(\text{poly})$

Beispiele für DSPACE: DSPACE(0)

- (nach unserer Definition der TM) Eingabe ist read-only, Ausgabe ist write-only,
- Arbeitsband darf nicht benutzt werden (Breite 0)
einzige Speichermöglichkeit ist Zustand
- das sind endliche Automaten mit Ein- und Ausgabe
- andere Namen dafür: finite (rational) transducer
- Bsp: Berechnung des Nachfolgers in Binärdarstellung

Beispiele für DSPACE: QBF \in PSPACE

- QBF = die Menge der wahren aussagenlogischen Formeln mit Quantoren, ohne freie Variablen
- Bsp: welche F_i sind in QBF?
 $F_1 \equiv \forall x \exists y : (x \leftrightarrow \neg y)$, $F_2 \equiv \exists x \forall y \exists z : (x \oplus y \oplus z)$
- Satz: QBF \in PSPACE
- Beweis: Auswertung der Formel rekursiv
z.B. $\text{eval}(\forall x : F) = \text{eval}(F[x := 0]) \wedge \text{eval}(F[x := 1])$
benutze Keller (auf Arbeitsband) für die Verwaltung der UP-Aufrufe, die Kellertiefe ist $\leq |F|$
- das Bsp zeigt: die Ressource Platz kann man *nachnutzen*
... und dabei sehr viel mehr Zeit verbrauchen

Beispiele: Spiele in PSPACE

- QBF für Formeln der Gestalt $\forall x_1 \exists y_1 \forall x_2 \exists y_2 \dots \forall x_n \exists y_n : P(x_1, y_1, x_2, y_2, \dots, x_n, y_n)$
als Zweipersonenspiel: für den ersten Zug von x gibt es einen Antwortzug für y , so daß für jeden ...
und $P(\dots)$ beschreibt die Gewinnbedingung
- viele andere Zweipersonenspiele \in PSPACE
genauer, das Entscheidungsproblem $\{s \mid \text{Position } s \text{ ist sicherer Gewinn für Spieler 2}\}$
wenn die Länge des Spiels polynomiell beschränkt ist

Bsp: Stefan Reisch, *HEX ist PSPACE-vollständig*, Acta Inf. 15(2)1981, 167–191

Beziehungen zwischen Platz und Zeit

- Satz: $\text{DTIME}(f) \subseteq \text{DSPACE}(f)$. Beweis: klar.
- Satz: $\text{DSPACE}(f) \subseteq \text{DTIME}(2^{O(f)})$
(dabei $2^{O(f)} := \bigcup_{c>0} (n \mapsto 2^{c \cdot f(n)})$)

- Beweis: sei $M \in \text{DSpace}_{\text{Alg}}(f)$.
wenn $M(x)$ mit $f(|x|)$ Platz akzeptiert,
dann sind alle Konfigurationen verschieden (sonst \perp),
es gibt $\leq |\Sigma|^{f(|x|)} = 2^{\log_2 |\Sigma| f(|x|)}$ Konfigurationen
also höchstens so viele Schritte.
- Folgerung: f berechenbar $\Rightarrow \text{DSpace}(f) \subseteq \text{REC}$
Beweis $\text{DSpace}(f) \subseteq \text{DTIME}(2^{O(f)}) \subseteq \text{REC}$
- Folgerung: $\text{DSpace}(\log) \subseteq P$, $\text{PSPACE} \subseteq \text{DEXPTIME}$

Reduktion, Vollständigkeit

- Def. \leq_P wie gehabt
- Satz: PSPACE ist abgeschlossen bzgl. \leq_P :
 $\forall L_1, L_2 : L_1 \leq_P L_2 \wedge L_2 \in \text{PSPACE} \Rightarrow L_1 \in \text{PSPACE}$
Beweis: Reduktionsfunktion f ist in polynomieller Zeit, also in polynomiellem Platz berechenbar.
- Def: L ist PSPACE -vollständig, gdw. $L \in \text{PSPACE} \wedge \forall L' \in \text{PSPACE} : L' \leq_P L$
- Satz: QBF ist PSPACE -vollständig.
Beweis (folgt)

Nichtdeterministischer Platz

- Def: $\text{NSpace}(f)$: Probleme lösbar durch nichtdeterministische TM mit Arbeitsband $\leq f(|\text{Eingabe}|)$
- Satz $\text{DSpace}(f) \subseteq \text{NSpace}(f)$. (ist trivial)
- Satz (Savitch) $\text{NSpace}(f) \subseteq \text{DSpace}(f^2)$.
Beweis (Idee): es ist $I(x) \rightarrow_M^{\leq 2^{c \cdot f(|x|)}} F$ zu entscheiden.
verwende UP mit Spez. $R(x, e, y) \iff x \rightarrow_M^{\leq 2^e} y$:
 $R(x, 0, y) := x \rightarrow_M^{0,1} = (x = y) \vee (x \rightarrow_M y)$
 $R(x, e + 1, y) := \bigvee_{h \in \text{Config}} (R(x, e, h) \wedge R(h, e, y)).$

mit Nachnutzung des Platzes. Kellertiefe ist $\leq f(|x|)$, jeder Eintrag im Keller ist $\leq f(|x|)$, Keller braucht Platz $\leq f(|x|)^2$

- Bemerkung: gilt nur für platzkonstruierbare $f \in \Omega(\log)$
- Folgerung $\text{DSPACE}(\text{poly}) = \text{NSPACE}(\text{poly}) = \text{PSPACE}$

QBF ist PSPACE-vollständig

- noch zu zeigen: $L \in \text{PSPACE} \Rightarrow L \leq_P \text{QBF}$.

- sei M eine PSPACE-Maschine für L ,

- benutze Formel von vorhin

$$R(x, e + 1, y) := \bigvee_{h \in \text{Config}} (R(x, e, h) \wedge R(h, e, y)).$$

- äquivalent:

$$R(x, e + 1, y) := \exists h \forall a, b : ((a = x \wedge b = h) \vee \dots) \Rightarrow R(a, e, b)$$

- damit erhält man eine Formel

der Größe $\leq \log_2(|\Sigma|^{\text{poly}(|x|)})$, also polynomiell in $|x|$

diese ist wahr gdw. M akzeptiert Eingabe x .

PSPACE-vollst. Einpersonenspiele

- $M \in \text{PSPACE}$ akzeptiert x

\iff es gibt eine passende Konfigurationsfolge.

- Platz für jede Konf. $\leq f(|x|)$ mit $f \in \text{poly}$, Länge $\leq 2^{O(f)}$

- Konf. kann auch die Anordnung von Spielsteinen auf einem (beschränkten) Brett sein

- Übergang = Spielzug = (z.B.) Verschieben von Steinen

- Joseph C. Culberson: *Sokoban is PSPACE-complete*, 1997, <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.41>

Sokoban $\in \text{PSPACE}$ ist leicht, Vollständigkeit ist schwer.

- Rush Hour ist PSPACE-vollständig. Lunar Lockout auch?

Zusammenfassung Zeit und Platz

- aus bisherigen Betrachtungen folgt:
 $L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP$
(dabei $L = DSPACE(\log)$, $EXP = DTIME(\exp)$)
- es ist anzunehmen, daß alle Inklusionen echt sind
aber zeigen kann man derzeit nur
 $L \subset PSPACE$, $P \subset EXP$
durch Platz- und Zeithierarchiesätze

Ein Hierarchie-Satz

- Def. zeit-beschränktes Halteproblem
 $K_f := \{(x, y) \mid \phi_x(y) \text{ akzeptiert in } \leq f(|y|) \text{ Schritten}\}$
- Satz: $K_f \in DTIME(f^3)$. Beweis: Simulation durch eine Maschine S mit zwei Bändern (eines für x , anderes als Arbeitsband für ϕ_x)
- Satz: $K_f \notin DTIME(f/2)$. Beweis (indirekt). Falls K_f entschieden durch M , dann betrachte Programm $P : x \mapsto \neg M(x, x)$ und Aufruf $P(P)$.
- Folgerung $DTIME(f) \subset DTIME((2f)^3)$.
- Folgerung $PTIME \subset EXPTIME$
Bew: K_f für $f = (n \mapsto 2^n)$.

20 Schaltkreise und Parallelität

Schaltkreis-Klassen

- AC^d, NC^d : Menge der Entscheidungsprobleme,
die lösbar sind durch Schaltkreise mit Operationen
 \neg sowie \wedge, \vee (zweistellig: NC^d , mehrstellig: AC^d)
uniform mit Größe $O(\text{poly})$, Tiefe $O(\log^d(n))$
- Satz: Parität $\in NC^1$, Parität $\notin AC^0$

- Bsp: Produkt von Relationen (Booleschen Matrizen) $\in \text{AC}^0$, auch $\in \text{NC}^1$
- Bsp: reflexiv-transitive Hülle $\in \text{AC}^1$, auch $\in \text{NC}^2$
- vgl. <https://complexityzoo.uwaterloo.ca/>

Beziehungen zu Zeit- und Platzklassen

- Satz: $\forall d \geq 0 : \text{NC}^d \subseteq \text{AC}^d \subseteq \text{NC}^{d+1}$
- Def: $\text{NC} := \bigcup_{d \geq 0} \text{NC}^d$, $\text{AC} := \bigcup_{d \geq 0} \text{AC}^d$.
- Satz: $\text{NL} \subseteq \text{NC} = \text{AC} \subseteq \text{P}$. Beweise:
 - $\text{NC} = \text{AC}$: wg. vorigem Satz
 - $\text{NL} \subseteq \text{AC}$: refl-tr. Hülle der Übergangsrelation (auf $\exp(c \cdot \log |x|) = \text{poly}(|x|)$ vielen Zuständen)
 - $\text{AC} \subseteq \text{P}$: Schaltkreis-Auswertung schichtweise

\leq_L und P-Vollständigkeit

- Def: $A \leq_L B$ falls $\exists f \in \text{FSPACE}(\log) : \forall x : x \in A \iff f(x) \in B$
- Satz: \leq_L ist transitiv.
Beweis — Vorsicht: $g(f(x))$ naiv benötigt zuviel Platz!
- Satz: $A \leq_L B \wedge B \in \text{NC} \Rightarrow A \in \text{NC}$
- Satz: $A \leq_L B \wedge B \in \text{P} \Rightarrow A \in \text{P}$
- Def: B heißt **P-vollständig**: $B \in \text{P} \wedge \forall A \in \text{P} : A \leq_L B$.
- Satz: Schaltkreisauswertung ist **P-vollständig**.
- Modellierung für parallele Komplexität:
 - **NC**: Aufgaben mit gut parallelisierbarer Lösung
 - \leq_L : Reduktion, die Parallelisierbarkeit erhält
 - **P-vollständig**: keine gute parallelen Alg. bekannt,

21 Plan der Vorlesung

Übersicht nach Kalenderwochen

- KW 46: Einführung, GOTO-Programme
- KW 47: WHILE-Programme, LOOP-Programme
- KW 48: universelle Programme, Halteproblem, Satz v. Rice
- KW 49: REC, RE, \leq_m
- KW 50: Turingmaschinen, Markov-Algorithmen, Wortersetzung
- KW 51: Unentscheidbare Eigensch. von Grammatiken, in der Logik
- KW 54: Nichtdeterminismus/Orakel, Komplexitätstheorie, P, NP
- KW 55: \leq_P , NP-vollständige Probleme
- KW 56: Zusammenfassung, Ausblick