192.017 Theoretische Informatik, 6.0 VU, 2023W

Chris FERMÜLLER	(chrisf@logic.at)
Lucas KLETZANDER	(lkletzan@dbai.tuwien.ac.at)
Marion OSWALD	(marion@logic.at)
Reinhard PICHLER	(pichler@dbai.tuwien.ac.at)
Gernot SALZER	(salzer@logic.at)
Stefan WOLTRAN	(woltran@dbai.tuwien.ac.at)

unter Mitwirkung von 25 Tutor_innen

thinf@list.tuwien.ac.at

Voraussetzung und Anmeldung

Voraussetzung

- abgeschlossene STEOP

Anmeldung

- Anmeldung im TISS (`tiss.tuwien.ac.at`) **bis 16.10.2023**

Vorlesungs- und Tutoriumstermine

Vorlesung

- Mo, 13.00 – 15.00, Informatikhörsaal
- Mi, 13.00 – 15.00, Informatikhörsaal

4 Tutorien

- jeweils vor den Abgabeterminen der 4 Übungsblätter
- 1. Tutorium: 18.10., 13.00 – 15.00, Informatikhörsaal
- weitere Tutorien: nach Ankündigung im TUWEL;
  voraussichtlich an eigenen Terminen und in anderen Hörsälen

Gliederung der Vorlesung

Teil 1 (Reinhard Pichler):
Einführung in Berechenbarkeit und Entscheidbarkeit

- Problem - Programm - Algorithmus
- Berechenbarkeit, Entscheidbarkeit
- Unentscheidbarkeit, Semi-Entscheidbarkeit

Teil 2a (Marion Oswald):
Formale Sprachen

- Turingmaschinen
- Grammatiken
- Chomsky-Hierarchie

Teil 2b (Chris Fermüller):
Berechenbarkeit Vertiefung

- Turingmaschinen vs. WHILE- und GOTO-Berechenbarkeit
- primitiv-rekursive und μ-rekursive Funktionen
- (weitere) unentscheidbare Probleme
- alternative Berechnungsmodelle

Teil 3 (Stefan Woltran):
Einführung in Komplexitätstheorie

- Komplextätsklassen P und NP
- Vollständigkeit
- weitere Komplexitätsklassen

Teil 4 (Gernot Salzer):
Formale Semantik von Programmen

Übungsteil - voraussichtliche Abgabetermine

Teil 1	23.10.2023
Teil 2	27.11.2023
Teil 3	13.12.2023
Teil 4	10.1.2024

Bemerkung: Die Übungsblätter werden voraussichtlich nicht gleich "groß" sein und daher auch nicht gleich viele Punkte bringen.

Übungsteil - Übungsabgaben

- Angaben rechtzeitig im TUWEL
- **WICHTIG:** Upload (*pdf*! *a4 Hochformat*! *leserlich*!) der Übungsbeispiele im TUWEL (**Termine beachten!**)
- Annotierte Lösungen im TUWEL, zusätzliches persönliches Feedback nach Anmeldung im TUWEL möglich
- Beurteilt werden vernünftige Lösungsversuche, Fehler sind erlaubt
- Diskussion ist gut, Abschreiben schlecht (führt zu Punkteabzug!)

Beurteilung

Komponenten

- Übungsteil: max. 40 Punkte
- schriftlicher Abschlusstest: max. 60 Punkte

Voraussetzung für eine positive Note:

- min. 30 Punkte auf den Abschlusstest
- min. 50 Punkte insgesamt

Voraussetzung für ein Zeugnis:

- Die Übungen sind im Prinzip freiwillig.
- Ein Zeugnis wird ausgestellt, wenn jemand bei zumindest einer Übungsabgabe oder schriftlichen Prüfung teilgenommen hat.

Beurteilung: Gesamtnote

Gesamtpunkte	Note
100 – 88	1
87 – 75	2
74 – 62	3
61 – 50	4
49 – 0	5

Beurteilung

Schriftliche Prüfung (Abschlusstest)

- keine Unterlagen außer **1 handgeschriebener DIN A4 Zettel**
- keine elektronischen Hilfsmittel (höchstens einfacher Taschenrechner)
- Haupttermin: Fr, 26.1.2024, 17:00-19:00, Anmeldung (TISS)!
- 3 Ersatztermine im Sommersemester
- Antritt bei **zwei von vier** Terminen möglich
- Pro (Stoff-)Semester wird **ein Zeugnis** ausgestellt
(Positive Zeugnisse werden sofort ausgestellt)
- **Spätestens nach dem 4. Termin wird ein Zeugnis ausgestellt!**
- **Übungsleistungen verfallen spätestens nach dem 4.Termin!**

Lehrziel und Inhalt

Ziel:

- (knowledge) Vermittlung von Grundbegriffen und Methoden der theoretischen Informatik (in Ergänzung und Vertiefung der in der LVA "Grundzüge digitaler Systeme STEOP" erworbenen Kenntnisse)
- (skills) einige zentrale Fertigkeiten in der (theoretischen) Informatik: Anwendung von Problemreduktionen, formale Argumentation

Inhalt:

- Berechenbarkeit, Entscheidbarkeit (Teil 1 und 2b)
- Formale Sprachen und Automaten (Teil 2a)
- Komplexitätstheorie (Teil 3)
- Programmverifikation, Hoare-Kalkül (Teil 4)

Unterlagen

- Vorlesungsfolien (TUWEL)
- Literatur(empfehlung) für ersten Teil:
  - J.E. Hopcroft, R. Motwani, J.D. Ullman: *Einführung in die Automatentheorie, Formale Sprachen und Berechenbarkeit.* (3., aktualisierte Auflage) Pearson Studium, 2011.
  - M. Sipser: *Introduction to the Theory of Computation.* (3rd edition) Cengage Learning, 2012.

Weitere Informationen

- Primäre Anlaufstelle für Fragen: `TUWEL` (Diskussionsforum)
- **`thinf@list.tuwien.ac.at`**

6.0 VU Theoretische Informatik (192.017)

Teil 1: Einführung in Berechenbarkeit und Entscheidbarkeit

Reinhard Pichler

Institut für Logic and Computation

Wintersemester 2023

Teil 1: Einführung in Berechenbarkeit und Entscheidbarkeit

Teil 1.1: Probleme, Programme, Algorithmen

Teil 1.2: Berechenbarkeit, Entscheidbarkeit

Teil 1.3: Semi-Entscheidbarkeit

Teil 1.4: Komplement

Teil 1.5: Jenseits der Semi-Entscheidbarkeit

Teil 1.6: Reduktionen

Probleme: Formale Definition

Definition von "Problemen"

Ein Problem ist definiert durch eine (abzählbar) unendliche Menge von möglichen Instanzen (d.h.: möglichen Inputs) zusammen mit einer Frage.

Ein Entscheidungsproblem ist ein Problem, bei dem die Frage eine ja/nein Antwort erwartet.

Beispiel für ein Problem

GRAPH-ERREICHBARKEIT:
INSTANZ: Ein Graph $G = (V, E)$ und Knoten $u, v \in V$.
FRAGE: Gibt es im Graph G einen Pfad von u nach v?

GRAPH-ERREICHBARKEIT ist ein Entscheidungsproblem.

Einige weitere Problemtypen

d.h.: andere Fragen als die mit ja/nein Antwort

- Funktionsproblem: Welches Ergebnis liefert die Anwendung einer Funktion f auf eine gegebene Instanz x (d.h.: die Anwort ist $f(x)$).
- Optimierungsproblem: berechne den optimalen Wert (= min/max) einer Funktion $f(x)$ (abhängig von der gegebenen Instanz x), z.B.: Länge des kürzesten Pfades zwischen 2 gegebenen Knoten.

Beobachtung: In der Informatik gibt es häufig den Begriff einer "Lösung". Die folgenden Problemtypen stellen dazu unterschiedliche Fragen.

- Suchproblem: Finde zu einer gegebenen Instanz *eine* Lösung, z.B.: finde einen konkreten Pfad von u nach v.
- Aufzählproblem: berechne alle "Lösungen" zu einer gegebenen Probleminstanz, z.B.: alle (zyklen-freien) Pfade von u nach v
- Zählproblem: berechne die Anzahl der "Lösungen"zu einer gegebenen Probleminstanz, z.B.: Wie viele unterschiedliche (zyklen-freie) Pfade von u nach v gibt es?

Motivation

Frage: Wieso werden in der Definition von “Problemen” unendlich viele Instanzen verlangt?

Antwort:

- Ein Lösungsverfahren muss auf beliebige Instanzen der Problemstellung anwendbar sein.
- Wenn es nur *endlich* viele Instanzen gäbe, wäre theoretisch ein Lösungsverfahren mit look-up in der Tabelle aller Instanzen samt Lösung möglich.
- *Unendlich* viele Instanzen stellen sicher, dass wir bestimmte mathematische Werkzeuge zur Analyse der Schwierigkeit von Problemen anwenden können.

Beispiele: Ein Lösungsverfahren für GRAPH-ERREICHBARKEIT muss auf *beliebige* Graphen anwendbar sein, eine SQL Anfrage muss für *beliebige* Daten zu gegebenem Schema gelten, etc.

Algorithmen

Definition eines Algorithmus

Ein Algorithmus für ein Problem $\mathcal{P}$ ist eine Beschreibung von Rechenschritten, die es uns erlauben, jede beliebige Instanz des Problems $\mathcal{P}$ zu lösen.

- Diese Definition ist ein bisschen vage ...
  - Was ist in eine zulässige "Beschreibung"?
  - Was ist ein zulässiger "Rechenschritt"?
- Natürliche Anforderungen: Ein Algorithmus muss auf alle Instanzen des Problems anwendbar sein und dabei ...
  1. nach endlich vielen Schritten terminieren und
  2. die korrekte Antwort auf die Frage liefern.
- Außerdem verlangen wir, dass
  3. jeder einzelne Rechenschritt "einfach" sein muss (d.h.: er kann von einer Maschine ausgeführt werden) und
  4. die Beschreibung auch für Menschen verständlich sein muss.

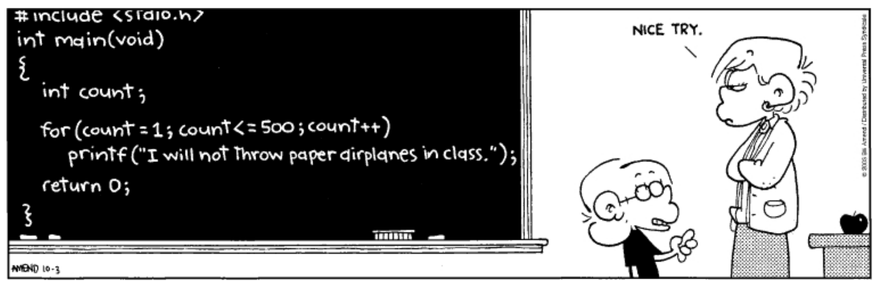
Berechenbarkeitstheorie

- **Zentrale Frage**:
  Welche Probleme lassen sich mit Hilfe eines Algorithmus lösen?
- Insbesondere: Was ist, wenn wir für ein Problem keinen Algorithmus finden? Ist es unsere Schuld oder gibt es für dieses Problem gar keinen Algorithmus? Wie können wir mit mathematischen Methoden beweisen, dass es für ein Problem keinen Algorithmus gibt?
- Gibt es überhaupt Probleme, für die kein Algorithmus existiert?
- Um mathematische Methoden anwenden zu können, brauchen wir ein Modell für den Berechenbarkeitsbegriff.
- Berechenbarkeitsmodell im ersten Teil der Vorlesung: eine einfache imperative Programmiersprache (einfache Datentypen, Variablen, Wertzuweisung, if-then-else, while, etc.)
- Im zweiten Teil der Vorlesung: einfachere Modelle, mit denen sich auch kompliziertere formale Beweise führen lassen.

Programme

Programmiersprachen

Programmiersprachen erlauben uns, Programme zu schreiben, die eine formale Beschreibung von Rechenschritten darstellen.



Programme

- In imperativen Programmiersprachen (wie C oder Java) werden die Rechenschritte explizit angegeben.
- In deklarativen Programmiersprachen (funktional wie HASKELL, logisch wie PROLOG oder auch SQL) werden die Rechenschritte implizit angegeben:
  - Man spezifiziert, wie die Lösung aussieht.
  - Ein Interpreter oder Compiler erzeugt daraus konkrete Rechenschritte, mit denen das gewünschte Ergebnis erzeugt wird.
- Programme sollen für Menschen verständlich / nachvollziehbar sein und von Computern ausgeführt werden können.

Algorithmen vs. Programme

Beobachtung: Nicht alle Programme sind gültige Algorithmen!

- Möglicherweise produziert ein Programm nicht für alle erlaubten Instanzen ein Antwort, z.B.: wegen Endlosschleife oder exception.
- Aber jedes Programm, das auf jeder Instanz terminiert, bildet einen Algorithmus für *ein* Problem.

Berechenbarkeit

- Die gewählte Programmiersprache bildet unser Modell für Berechenbarkeit.
- Damit können wir eine zentrale Frage beantworten: Gibt es für ein gegebenes Problem einen Algorithmus (= ein Programm, das garantiert terminiert und die korrekte Antwort liefert)?
- Offensichtlich: für viele Probleme gibt es einen Algorithmus.
- Weniger offensichtlich: für etliche natürliche Probleme gibt es keinen Algorithmus (wesentlicher Teil dieser Vorlesung)

Unser Programmiersprache: SIMPLE

- Wir verwenden den Kern von prozeduralen Programmiersprachen:
  - Variablen von unterschiedlichem Typ (String, Integer, Boolean, Void)
  - Wertzuweisungen, einfache Rechenoperationen (z.B.: $x := y + z$)
  - "if/then/else" Anweisungen
  - "while" Schleifen
  - "for" und "repeat" Schleifen
    (können durch "while" Schleifen ersetzt werden)
  - "return" Anweisungen
  - Prozeduren, Funktionen
- Der Input für ein Programm kann sein:
  - eine Liste $L = (V_1, V_2, \ldots, V_n)$ von Werten unterschiedlicher Typen;
  - oder einfach ein (großer) String I: Jede Liste L lässt sich als String über einem beliebigen "Alphabet" darstellen (z.B.: ASCII, $\{0, 1\}$, etc.)
- Ein Programm liefert als Ergebnis einen Wert (von beliebigem Typ) mittels "return" Anweisung.

Unsere Programmiersprache (Beispiel)

Problem: Primzahltest

INSTANZ: natürliche Zahl n;
FRAGE: Ist n eine Primzahl?

Programm (Algorithmus)

```
Boolean isPrime(Integer n)
    if n < 2 then return false;
    i:=2;
    while i < n do {
        if n % i = 0 then return false;
        i := i + 1; }
    return true;
```

Angemessenheit unserer Programmiersprache SIMPLE

- Ist die Programmiersprache SIMPLE ein geeignetes Modell für Berechenbarkeit?
  - d.h.: Lassen sich in SIMPLE *alle* Algorithmen ausdrücken?
  - Oder gibt es Probleme, für die es zwar ein Programm in Java, C oder einer anderen Programmiersprache aber kein SIMPLE Programm gibt?

- Antwort: SIMPLE ist ein geeignetes Modell, d.h. alle Probleme, die in JAVA, C oder einer anderen bekannten Programmiersprache gelöst werden können, lassen sich auch in SIMPLE lösen.
- Begründung:
  - In SIMPLE lassen sich Interpreter oder Compiler für alle bekannten Programmiersprachen implementieren, z.B.:
  - eine Java Virtual Machine (JVM), um JAVA Programme auszuführen.

- Konsequenz: Wenn wir beweisen können, dass es für ein Problem kein SIMPLE Programm gibt, dann gibt es generell keinen Algorithmus für dieses Problem (auch nicht in einer anderen Programmiersprache).

Teil 1: Einführung in Berechenbarkeit und Entscheidbarkeit

- Teil 1.1: Probleme, Programme, Algorithmen
- Teil 1.2: Berechenbarkeit, Entscheidbarkeit
- Teil 1.3: Semi-Entscheidbarkeit
- Teil 1.4: Komplement
- Teil 1.5: Jenseits der Semi-Entscheidbarkeit
- Teil 1.6: Reduktionen

Berechenbarkeit

Berechenbarkeit von Funktionen

Wir nennen eine Funktion berechenbar, wenn es einen Algorithmus gibt, der ein beliebiges Element x der Grundmenge von f als Input nimmt und als Output den Funktionswert $f(x)$ liefert.

Bemerkung:

- Im Prinzip kann die Grundmenge aus beliebigen Objekten bestehen, z.B.: Zahlen, Strings, Graphen, Datenbanken (möglicherweise eingeschränkt durch ein Schema), etc.
- Die Objekte müssen sich als (endliche) Strings über einem geeigneten (endlichen) Alphabet codieren lassen.
- wichtige Arten von hier betrachteten Funktionen:
  $f : \mathbb{N} \to \mathbb{N}$, $f : \Sigma^* \to \Sigma^*$, $f : \mathbb{N} \to \{0, 1\}$, $f : \Sigma^* \to \{0, 1\}$.
- Statt $\{0, 1\}$ kann man auch $\{$nein,ja$\}$ oder $\{$falsch,wahr$\}$ nehmen, d.h.: das *Funktionsproblem* ist eigentlich ein *Entscheidungsproblem*.

Entscheidbarkeit

Entscheidbarkeit

Wie nennen ein Entscheidungsproblem $\mathcal{P}$ entscheidbar, wenn es einen Algorithmus gibt, der eine beliebige Instanz x des Problems $\mathcal{P}$ als Input nimmt und die korrekte Antwort ja / nein (oder 1 / 0 oder wahr / falsch) als Output liefert.

Bemerkung:

- Unser Modell für Algorithmen sind SIMPLE Programme.
- d.h.: Eine Funktion ist berechenbar bzw. ein Entscheidungsproblem ist entscheidbar, wenn es dafür ein SIMPLE Programm gibt.
- offensichtlich: viele Funktionen sind berechenbar und viele Entscheidungsprobleme sind entscheidbar (siehe z.B.: AlgoDat).
- weniger offensichtlich: es gibt auch nicht-berechenbare Funktionen und nicht-entscheidbare Entscheidungsprobleme.

Existenz von unentscheidbaren Problemen

Beobachtung

- SIMPLE Programme sind endliche Strings über einem endlichen Alphabet Σ (z.B. 127 ASCII-Zeichen).
- Auch die Instanzen eines Entscheidungsproblems können als Strings über Σ dargestellt werden.
- Entscheidungsprobleme entsprechen dann Funktionen $\Sigma^* \to \{0, 1\}$.

Theorem

- *Die Menge Σ^* ist abzählbar unendlich, d.h.: es existiert eine bijektive Funktion $g : \Sigma^* \to \mathbb{N}$.*
- *Die Menge der Funktionen $\mathbb{N} \to \{0, 1\}$ ist überabzählbar (und daher ist auch die Menge der Funktionen $\Sigma^* \to \{0, 1\}$ überabzählbar).*

Existenz von unentscheidbaren Problemen

Zusammenfassung

- Es gibt also überabzählbar viele Funktionen und Entscheidungsprobleme.
- Es gibt aber nur abzählbar viele endliche Strings über einem endlichen Alphabet. Daher gibt es nur abzählbar viele SIMPLE Programme.
- d.h.: es muss Funktionen und Entscheidungsprobleme geben, für die es kein SIMPLE Programm (und somit keinen Algorithmus) gibt.

Was wir noch zeigen müssen:

- Abzählbarkeit von Σ^*
- Überabzählbarkeit von $\{0,1\}^{\mathbb{N}}$
  (d.h.: Menge der Funktionen $f : \mathbb{N} \to \{0,1\}$)

Abzählbarkeit von Σ^*

Beweisidee:

Angenommen $\Sigma = \{a_1, \ldots, a_k\}$. Dann können wir die Menge Σ^* auf folgende Weise aufzählen:

- Aufzählung der Strings geordnet nach der Länge
- Anordnung innerhalb der selben Länge: lexikographisch

Eine bijektive Abbildung $g \colon \Sigma^* \to \mathbb{N}$ erhält man, indem man jedes Element in der Aufzählung auf seine Position in der Aufzählung abbildet.

Aufzählung:

Strings	Anzahl	Positionen
ϵ (leerer String)	1	0
$a_1, \ldots, a_k$	k	$1, \ldots, k$
$a_1 a_1, a_1 a_2, \ldots, a_k a_k$	k^2	$k + 1, \ldots, k^2 + k$
$a_1 a_1 a_1, a_1 a_1 a_2, \ldots, a_k a_k a_k$	k^3	$k^2 + k + 1, \ldots, k^3 + k^2 + k$
etc.		

Überabzählbarkeit von $\{0,1\}^N$

Beweisidee (Cantor'sches Diagonalverfahren) :

- Indirekt: angenommen, die Funktionen in $\{0,1\}^N$ sind abzählbar. Dann lassen sie sich (so wie die Elemente in Σ^*) aufzählen.
- Jede Funktion lässt sich als Folge von Nullen und Einsern darstellen.
- Indem wir entlang der “Diagonale” jeweils den Wert “umdrehen”, erzeugen wir eine Zahl, die in der Aufzählung nicht vorkommt.

Aufzählung von $\{0,1\}^{\mathbb{N}}$:

$$
\begin{array}{cccccccc}
f_0 & = & \mathbf{a_{00}} & a_{01} & a_{02} & a_{03} & a_{04} & \dots \\
f_1 & = & a_{10} & \mathbf{a_{11}} & a_{12} & a_{13} & a_{14} & \dots \\
f_2 & = & a_{20} & a_{21} & \mathbf{a_{22}} & a_{23} & a_{24} & \dots \\
\vdots & = & \vdots & \vdots & \vdots & \vdots & \vdots &
\end{array}
$$

Definiere nun eine Funktion $\hat{f} : \mathbb{N} \to \{0,1\}$ mit $\hat{f}(n) = 1 - f_n(n)$.
Diese Funktion ist in der Aufzählung nicht enthalten. Widerspruch!

Bemerkung: Cantor hat so die Überabzählbarkeit von $\mathbb{R}$ gezeigt.

Probleme über Programme

- Es gibt viele natürliche Probleme, für die ein Algorithmus nicht offensichtlich ist! Dazu gehören insbesondere zahlreiche Probleme, die Fragen zum Verhalten von Programmen betreffen, z.B.:
  - Gegeben ein Programm Π und ein Input I, tritt das Programm irgendwann in eine Endlosschleife?
  - Gegeben ein Programm Π, terminiert es auf *allen* Inputs?
- Algorithmen für solche Probleme wären sehr hilfreich, um die Korrektheit von Programmen sicherzustellen.
- Außerdem könnten einige (zum Teil "klassische") mathematische Probleme gelöst werden, wenn wir einen Algorithmus zum Entscheiden der Programm-Termination hätten, z.B.:
  - Goldbachsche Vermutung, die eines der berühmtesten offenen Probleme der Mathematik darstellt (aufgestellt 1742 von Christian Goldbach in einem Brief an Leonhard Euler).

Goldbachsche Vermutung

Jede gerade Zahl größer als 2 ist die Summe von 2 Primzahlen.

Wir können ein Programm schreiben, das diese Aussage für $4, 6, 8, 10, 12, \ldots$ überprüft.

```
Boolean test(Integer n)    /* checks if n is the sum of two primes */
    for all i ≤ n, j ≤ n do {
        if (isPrime(i) and isPrime(j) and i + j = n) then return true; }
    return false;

Void testConjecture()
    n:=4;
    while test(n)=true do { n := n + 2; }
```

Theorem

Die Goldbachsche Vermutung ist wahr $\Leftrightarrow$
testConjecture() terminiert nicht.

Versuch eines computergestützten Beweises

Idee

- Angenommen wir hätten ein Programm Π_h, das entscheidet, ob das Programm testConjecture() terminiert.
- Indem wir das Programm Π_h auf dem Computer laufen lassen, könnten wir die Gültigkeit der Goldbachschen Vermutung entscheiden.
  - Falls Π_h “nein” ausgibt, ist die Vermutung bewiesen.
  - Falls Π_h “ja” ausgibt, ist die Vermutung widerlegt, d.h., es gibt eine gerade Zahl $n > 2$, die nicht die Summe von 2 Primzahlen ist.

Bemerkung:

- Wir wissen nicht, wie lange die Ausführung von Π_h dauert. Aber da es ein Algorithmus ist, terminiert Π_h garantiert und löst somit die Goldbachsche Vermutung.
- Zahlreiche andere mathemtatische Probleme könnten mittels Algorithmus zum Entscheiden der Programm-Termination gelöst werden, z.B.: der große Fermatsche Satz.

Unentscheidbarkeit des HALTEPROBLEMS

HALTEPROBLEM

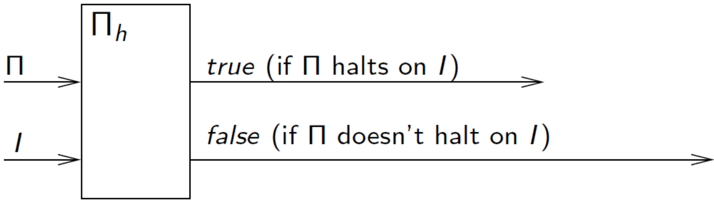
INSTANZ: (Quellcode von einem) Programm Π, Input String I.
FRAGE: Terminiert das Programm Π auf dem Input I?

- Wir beweisen, dass es keinen Algorithmus (also kein terminierendes Programm Π_h) gibt, das das HALTEPROBLEM entscheidet.
- Der Beweis ist indirekt, d.h.: wir nehmen an, dass das HALTEPROBLEM mit Hilfe eines Programms Π_h entscheidbar ist und zeigen, dass diese Annahme zu einem Widerspruch führt.
- Wir nehmen an, dass sowohl der Quellcode des Programms Π als auch der Input I für Π als String gegeben ist.

Unentscheidbarkeit des HALTEPROBLEMS (Schritt 1)

Wie nehmen an, dass es ein Programm Π_h mit folgender Eigenschaft gibt:

- Π_h nimmt 2 Strings als Input:
  - Π (Quellcode eines SIMPLE Programms)
  - I (Input für das Programm Π)
- Π_h erzeugt folgenden Output:
  - *true* falls das Programm Π auf Input I terminiert;
  - *false* falls das Programm Π auf Input I *nicht* terminiert.

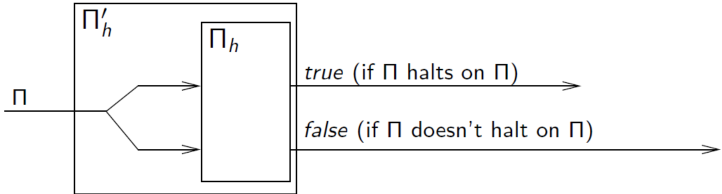


Boolean Π_h (String Π, String I)
  /* code to check if Π terminates on I. */

Unentscheidbarkeit des HALTEPROBLEMS (Schritt 2)

Mit Hilfe von Π_h erzeugen wir nun ein Programm Π'_h:

- Π'_h nimmt als Input *einen* String, dupliziert diesen und ruft mit diesen 2 Strings das Programm Π_h auf.
- d.h.: Π'_h prüft, ob ein Program Π terminiert, wenn es seinen eigenen Quellcode als Input nimmt.

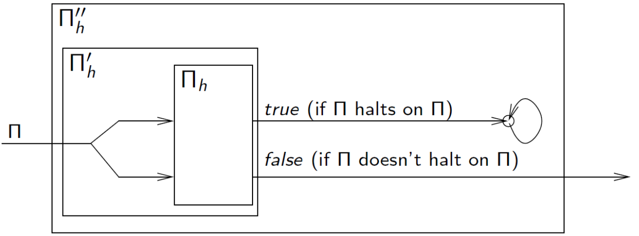


Boolean Π'_h (String Π)
return $\Pi_h(\Pi, \Pi)$;

Unentscheidbarkeit des HALTEPROBLEMS (Schritt 3)

Mit Hilfe von Π'_h erzeugen wir nun ein weiteres Programm Π''_h (und zwar verändern wir das Output-Verhalten von Π'_h):

- Falls Π'_h *true* liefert, dann geht Π''_h in eine Endlosschleife!



Boolean Π''_h (String Π)
  if $\Pi'_h(\Pi) = true$ **then** { **while** (true) **do** {} }
    else return *false*;

Unentscheidbarkeit des HALTEPROBLEMS (Schritt 4)

Beobachtung:

1. Falls Π auf dem Input Π hält, dann hält Π''_h auf dem Input Π *nicht*.
2. Falls Π auf dem Input Π *nicht* hält, dann hält Π''_h auf dem Input Π.

Was passiert, wenn Π''_h mit (dem Quellcode von) Π''_h als Input läuft?

Es kann nur zwei Möglichkeiten geben:

1. Fall 1: Π''_h hält auf dem Input Π''_h. Dann folgt aus Beobachtung 1, dass Π''_h *nicht* auf dem Input Π''_h hält. Widerspruch!
2. Fall 2: Π''_h hält *nicht* auf dem Input Π''_h. Dann folgt aus Beobachtung 2, dass Π''_h auf dem Input Π''_h hält. Widerspruch!

d.h.: unsere ursprüngliche Annahme (nämlich, dass es ein Programm Π_h gibt, das das HALTEPROBLEM entscheidet) war falsch und es gilt:

Theorem

Das HALTEPROBLEM *ist unentscheidbar.*

Weitere Beispiele von unentscheidbaren Problemen

KORREKTHEIT

INSTANZ: (Quellcode von einem) Programm Π und 2 Strings I_1, I_2.
FRAGE: Liefert Π bei Aufruf mit Input I_1 als Output den String I_2?

Intuition: KORREKTHEIT ist unentscheidbar, weil die Beantwortung dieser Frage implizit die Frage beantworten müsste, ob Π auf Input I_1 hält.

ERREICHBARER-CODE

INSTANZ: (Quellcode von einem) Programm Π, eine natürliche Zahl n.
FRAGE: Gibt es einen Input I für das Programm Π, sodass Π bei Ausführung mit Input I den Code auf der Programmzeile n ausführt?

Optimierungspotenzial: unerreichbarer Code kann gelöscht werden.

Intuition: ERREICHBARER-CODE ist unentscheidbar, weil wir das HALTEPROBLEM erhalten, wenn wir die *letzte* Zeile in Π als n nehmen.

Bemerkung: Diese intuitiven Argumente für Unentscheidbarkeit werden wir formalisieren, wenn wir den Begriff der "Reduktion" einführen.

Teil 1: Einführung in Berechenbarkeit und Entscheidbarkeit

Teil 1.1: Probleme, Programme, Algorithmen

Teil 1.2: Berechenbarkeit, Entscheidbarkeit

Teil 1.3: Semi-Entscheidbarkeit

Teil 1.4: Komplement

Teil 1.5: Jenseits der Semi-Entscheidbarkeit

Teil 1.6: Reduktionen

Semi-entscheidbare Probleme

Wir lockern die Bedingung für Entscheidbarkeit und definieren semi-entscheidbare Probleme:

Definition

Wir nennen ein Entscheidungsproblem $\mathcal{P}$ semi-entscheidbar, wenn es ein Programm Π mit folgenden Eigenschaften gibt:

- Π nimmt eine Instanz I von $\mathcal{P}$ als Input;
- Falls I eine "ja" Instanz ist, dann liefert Π das Ergebnis *true*;
- falls I eine "nein" Instanz, dann kann Π entweder das Ergebnis *false* liefern oder nicht terminieren.

In anderen Worten:

- Π arbeitet korrekt auf allen positiven Instanzen von $\mathcal{P}$;
- Π darf auf negativen Instanzen von $\mathcal{P}$ endlos laufen;
- aber wenn Π auf einer negativen Instanz terminiert, dann muss Π das korrekte Ergebnis ($=$ *false*) liefern.

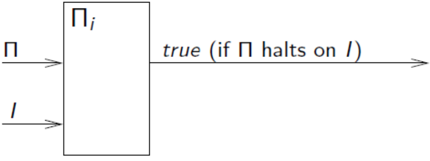
Noch einmal das HALTEPROBLEM

Theorem

Das HALTEPROBLEM *ist semi-entscheidbar.*

Beweisidee: Wir können ein Interpreter Programm Π_i mit folgenden Eigenschaften schreiben:

- Π_i nimmt als Input eine beliebige Instanz des HALTEPROBLEMS, d.h.: Quellcode von einem Programm Π und Input I für Π;
- Π_i analysiert Π und simuliert die Ausführung von Π auf Input I;
- Wenn die Simulation von Π auf I terminiert, gibt Π_i *true* aus;
- Wenn die Simulation von Π auf I nicht terminiert, dann läuft auch Π_i auf dem Input (Π, I) endlos.



Weitere semi-entscheidbare Probleme

Wiederholung: KORREKTHEIT

INSTANZ: (Quellcode von einem) Programm Π und 2 Strings I_1, I_2.
FRAGE: Liefert Π bei Aufruf mit Input I_1 als Output den String I_2?

Theorem

Das KORREKTHEIT*-Problem ist semi-entscheidbar.*

Beweisidee (analog zum HALTEPROBLEM): Wir können ein Interpreter Programm Π_i mit folgenden Eigenschaften schreiben:

- Π_i nimmt als Input eine beliebige Instanz vom KORREKTHEIT d.h.: Quellcode von einem Programm Π und zwei Strings I_1, I_2;
- Π_i analysiert Π und simuliert die Ausführung von Π auf Input I_1;
- Wenn die Simulation mit Output O terminiert, dann überprüft Π_i, ob $I_2 = O$ gilt. Wenn ja, dann gibt Π_i *true* aus, sonst *false*;
- Wenn die Simulation von Π auf I_1 nicht terminiert, dann läuft auch Π_i auf dem Input (Π, I) endlos.

Weitere semi-entscheidbare Probleme

Wiederholung: ERREICHBARER-CODE

INSTANZ: (Quellcode von einem) Programm Π, eine natürliche Zahl n.
FRAGE: Gibt es einen Input I für das Programm Π, sodass Π bei Ausführung mit Input I den Code auf der Programmzeile n ausführt?

Beobachtungen: Für ein gegebenes Paar (I, i), wobei I ein möglicher Input für Π ist und $i \in \mathbb{N}$, können wir

- einen Interpreter ausführen, der die ersten i Schritte von Π auf Input I simuliert, und
- der entscheidet, ob innerhalb der ersten i Schritte die Zeile n im Programm Π erreicht wurde.

Idee des Semi-Entscheidungsverfahrens:

- Alle Paare (I, i) aufzählen und überprüfen, ob Π bei Input I die Programmzeile n innerhalb von i Schritten erreicht.
- Falls (Π, n) eine positive Instanz ist, werden wir *garantiert irgendwann* so ein Paar (I, i) finden und geben *true* aus.
- Falls (Π, n) eine negative Instanz ist, läuft die Aufzählung endlos.

Einschub: Aufzählung, Abzählbarkeit

Frage: Wie können wir in einem Semi-Entscheidungsverfahren für das KORREKTHEIT-Problem alle Paare (I, i) aufzählen?

Vorsicht: Eine geschachtelte Schleife funktioniert nicht, z.B.:

for all input $I \in \Sigma^*$
  for all $i \in \mathbb{N}$
    do simuliere die ersten i Programmschritte von Π auf Input I;

Begründung: Wenn für einen bestimmten Input I das Programm Π nicht terminiert, werden nach I keine weiteren Inputs mehr erzeugt, d.h.: wir testen also nicht alle Paare (I, i).

Allgemeine Problemstellung:

- Wie kann man $M_1 \times M_2$ aufzählen für zwei *abzählbar unendliche* Mengen M_1, M_2?
- Äquivalente Frage: Ist $M_1 \times M_2$ abzählbar, wenn die zwei Mengen M_1, M_2 *abzählbar unendlich* sind?

Abzählbarkeit (Fortsetzung)

Definition

Eine Menge M heißt abzählbar, wenn sie entweder endlich ist oder wenn es eine bijektive Abbildung $M \to \mathbb{N}$ gibt.

Beobachtungen:

- Falls M unendlich ist, müssen wir also eine bijektive Abbildung $f: M \to \mathbb{N}$ finden, um die Abzählbarkeit von M zu beweisen.
- Einfacher: definiere bijektive Abbildung $g: \mathbb{N} \to M$ als Aufzählung.
- Alternative: es reicht, eine *injektive* Abbildung $f: M \to \mathbb{N}$ zu finden.
- Intuition: Wenn es eine injektive Abbildung $f: M \to \mathbb{N}$ gibt, dann gilt offensichtlich $|M| \leq |\mathbb{N}|$.
- Formale Begründung: Für injektive Abbildung $f: M \to \mathbb{N}$ definieren wir folgende Abbildung g:

  $$g: M \to \mathbb{N} \text{ mit } g(a) = |\{i \mid i < f(a)\}| \text{ für alle } a \in M$$

  Man sieht leicht, dass g bijektiv (d.h. injektiv und surjektiv) ist.

Einige abzählbar unendliche Mengen

- $M = \mathbb{N} \cup \{a\}$ für beliebiges Objekt a:
  $f : M \to \mathbb{N}$ mit $f(a) = 0$ und $f(n) = n + 1$ für alle $n \in \mathbb{N}$ ist injektiv.
- $M = \mathbb{N} \cup \{a_1, \ldots, a_k\}$ für $k \geq 1$ und beliebige Objekte $a_1, \ldots, a_k$:
  $f : M \to \mathbb{N}$ mit $f(a_i) = i - 1$ und $f(n) = n + k$ für alle $n \in \mathbb{N}$
- $M = \{0, 1\} \times \mathbb{N}$:
  $f : M \to \mathbb{N}$ mit $f(i, n) = 2n + i$ für alle $i \in \{0, 1\}$ und $n \in \mathbb{N}$.
- $M = \mathbb{N} \times \mathbb{N}$ oder, allgemein:
  $M = M_1 \times M_2$ für abzählbar unendliche Mengen M_1, M_2.
  Idee: für $M = \mathbb{N} \times \mathbb{N}$:
  Aufzählung (Cantor'sches Abzählprinzip):
  (0,0),
  (1,0), (1,1), (0,1),
  (2,0), (2,1), (2,2), (1,2), (0,2)
  (3,0), (3,1), (3,2), (3,3), (2,3), (1,3), (0,3),
  etc.

Cantor'sches Abzählprinzip

Aufzählung von $M_1 \times M_2$ mit $M_1 = \{a_1, a_2, a_3, a_4, \ldots\}$ und $M_2 = \{b_1, b_2, b_3, b_4, \ldots\}$:

$$\begin{pmatrix} (a_1, b_1) & (a_1, b_2) & (a_1, b_3) & \ldots, \\ (a_2, b_1) & (a_2, b_2) & (a_2, b_3) & \ldots, \\ (a_3, b_1) & (a_3, b_2) & (a_3, b_3) & \ldots, \\ (a_4, b_1) & (a_4, b_2) & (a_4, b_3) & \ldots, \\ \vdots & \vdots & \vdots & \end{pmatrix} \quad \begin{pmatrix} 1 & 4 & 9 & 16 & \ldots, \\ 2 & 3 & 8 & 15 & \ldots, \\ 5 & 6 & 7 & 14 & \ldots, \\ 10 & 11 & 12 & 13 & \ldots, \\ \vdots & \vdots & \vdots & \vdots & \end{pmatrix}$$

Daraus folgt:

Theorem

Das Erreichbarer-Code *Problem ist semi-entscheidbar.*

Bemerkung: Cantor hat so die Abzählbarkeit von $\mathbb{Q}$ gezeigt.

Weitere semi-entscheidbare Probleme

Das "Entscheidungsproblem"

Instanz: eine Formel ϕ der Prädikatenlogik erster Stufe.
Frage: Ist ϕ gültig?

Theorem

Das Entscheidungsproblem *ist semi-entscheidbar.*

Wiederholung: Prädikatenlogische Formeln erster Stufe sind so definiert, dass man zuerst induktiv Terme definiert und damit dann ebenfalls induktiv Formeln:

Terme können sein:

- Konstantensymbole (üblicherweise $a, b, c, \ldots$)
- Variablen (üblicherweise x, y, z)
- zusammengesetzte Terme der Form $f(t_1, \ldots, t_\alpha)$ für ein Funktionssymbol f mit Arität α und Terme $t_1, \ldots, t_\alpha$.

Das Entscheidungsproblem

Formeln können sein:

- Atome der Form $P(t_1, \ldots, t_\alpha)$ für ein Prädikatensymbol P mit Arität α und Terme $t_1, \ldots, t_\alpha$
- oder zusammengesetzte Formeln der Form $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $(\exists x)\phi$ oder $(\forall x)\phi$ für Formeln ϕ, ψ und Variable x.

Geschichte:

- Das Entscheidungsproblem (der Prädikatenlogik erster Stufe) wurde bereits von Gottfried Leibniz im 17. Jahrhundert formuliert.
- Anfang des 20. Jahrhunderts von David Hilbert und Wilhelm Ackermann neu auf die "Tagesordnung" der Mathematik gesetzt.
- Semi-Entscheidbarkeit: aufgrund des "Vollständigkeitssatzes" von Kurt Gödel.
- Unentscheidbarkeit: unabhängig voneinander von Alonzo Church 1935 (mit λ-calculus als Modell für Berechenbarkeit) und Alan Turing 1936 (mit Turing Maschinen als Modell für Berechenbarkeit) bewiesen.

Das Entscheidungsproblem

Idee eines Unentscheidbarkeitsbeweises:

- Mittels Reduktion vom Halteproblem für Turing Maschinen.
- Reduktionen werden noch im Lauf von Teil 1 der LVA eingeführt; Turing Maschinen im Teil 2a.

Idee eines Semi-Entscheidungsverfahrens:

- Es gibt vollständige Kalküle für die Prädikatenlogik erster Stufe, d.h.: in einem vollständigen Kalkül gibt es für jede gültige Formel eine endliche Ableitung.
- z.B.: Hilbert Kalkül:
  - einige Axiome wie $A \rightarrow (B \rightarrow A)$, $(\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A)$, $[A \rightarrow (B \rightarrow C)] \rightarrow [(A \rightarrow B) \rightarrow (A \rightarrow C)]$, etc.
  - und Modus Ponens: wenn man bereits Formeln A und $(A \rightarrow B)$ abgeleitet hat, dann kann man daraus B ableiten.
- Semi-Entscheidungsverfahren: einfach alle möglichen Ableitungen in einem vollständigen Kalkül aufzählen.
  - Wenn die Formel ϕ gültig ist, wird man sie irgendwann ableiten.
  - Bei ungültigem ϕ werden endlos neue Formeln abgeleitet.

Teil 1: Einführung in Berechenbarkeit und Entscheidbarkeit

Teil 1.1: Probleme, Programme, Algorithmen

Teil 1.2: Berechenbarkeit, Entscheidbarkeit

Teil 1.3: Semi-Entscheidbarkeit

Teil 1.4: Komplement

Teil 1.5: Jenseits der Semi-Entscheidbarkeit

Teil 1.6: Reduktionen

Komplement eines Entscheidungsproblems

- Sei $\mathcal{P}$ ein Entscheidungsproblem, d.h.: eine unendliche Menge von möglichen Instanzen plus ja/nein-Frage.
- Das Komplement von $\mathcal{P}$ (oder "co-Problem von $\mathcal{P}$") besteht aus der selben Menge von Instanzen wie $\mathcal{P}$ aber mit verneinter Frage.
- Das co-Problem von $\mathcal{P}$ wird üblicherweise als co-$\mathcal{P}$ bezeichnet.
- Das co-Problem von co-$\mathcal{P}$ ist $\mathcal{P}$ (wegen doppelter Verneinung).

Wiederholung: GRAPH-ERREICHBARKEIT

INSTANZ: Ein Graph $G = (V, E)$ und Knoten $u, v \in V$.
FRAGE: Gibt es im Graph G einen Pfad von u nach v?

Das Komplement von GRAPH-ERREICHBARKEIT ist folgendes Problem:

NICHT-ERREICHBARKEIT (oder CO-ERREICHBARKEIT)

INSTANZ: Ein Graph $G = (V, E)$ und Knoten $u, v \in V$.
FRAGE: Gibt es im Graph G keinen Pfad von u nach v?

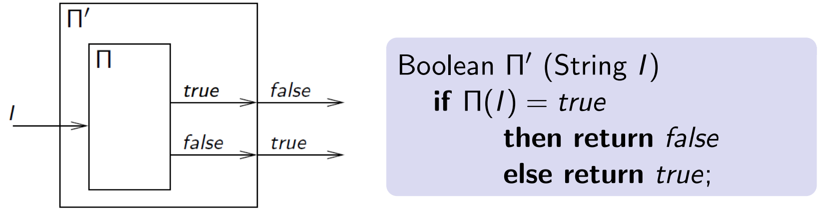
Eigenschaften der Komplementierung

Theorem

Wenn ein Problem $\mathcal{P}$ entscheidbar ist, dann ist auch sein Komplement co-$\mathcal{P}$ entscheidbar

Beweis:

- Wenn $\mathcal{P}$ entscheidbar ist, dann gibt es ein Programm Π, das für alle positiven Instanzen von $\mathcal{P}$ *true* und für alle negativen Instanzen von $\mathcal{P}$ *false* zurückgibt.
- Wir können Π zu einem neuen Programm Π' ändern, indem wir einfach den Ausgabewert invertieren.
- Dann ist Π' eine Entscheidungsprozedur für co-$\mathcal{P}$.

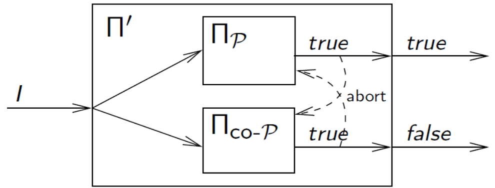


Theorem

Sei $\mathcal{P}$ ein Entscheidungsproblem mit Komplement co-$\mathcal{P}$. Wenn sowohl $\mathcal{P}$ als auch co-$\mathcal{P}$ semi-entscheidbar ist, dann ist $\mathcal{P}$ entscheidbar.

Beweis:

- Da $\mathcal{P}$ semi-entscheidbar ist, gibt es ein Programm $\Pi_{\mathcal{P}}$, das auf den positiven Instanzen von $\mathcal{P}$ terminiert und *true* ausgibt.
- Da co-$\mathcal{P}$ semi-entscheidbar ist, gibt es ein Programm $\Pi_{\text{co-}\mathcal{P}}$, das auf den positiven Instanzen von co-$\mathcal{P}$ (d.h.: auf den negativen Instanzen von $\mathcal{P}$) terminiert und *true* ausgibt.
- Wir definieren ein Programm Π', das $\Pi_{\mathcal{P}}$ and $\Pi_{\text{co-}\mathcal{P}}$ parallel ausführt. Eines der beiden Programme terminiert garantiert. Dann liefert Π' den entsprechenden Output und stoppt das andere Programm.



Teil 1: Einführung in Berechenbarkeit und Entscheidbarkeit

Teil 1.1: Probleme, Programme, Algorithmen

Teil 1.2: Berechenbarkeit, Entscheidbarkeit

Teil 1.3: Semi-Entscheidbarkeit

Teil 1.4: Komplement

Teil 1.5: Jenseits der Semi-Entscheidbarkeit

Teil 1.6: Reduktionen

Jenseits der Semi-Entscheidbarkeit

- Wir haben Probleme gesehen, die unentscheidbar aber semi-entscheidbar sind.
- Frage: Gibt es Probleme, die nicht einmal semi-entscheidbar sind?

Theorem

Sei $\mathcal{P}$ ein Entscheidungsproblem, so dass (1) $\mathcal{P}$ unentscheidbar ist und (2) $\mathcal{P}$ semi-entscheidbar ist. Dann ist co-$\mathcal{P}$ nicht semi-entscheidbar.

Beweis.

Indirekt: Angenommen, co-$\mathcal{P}$ ist semi-entscheidbar. Das heißt. sowohl $\mathcal{P}$ als auch co-$\mathcal{P}$ ist semi-entscheidbar. Dann folgt aus dem vorigen Theorem, dass $\mathcal{P}$ entscheidbar ist. Widerspruch zur Annahme (1). □

Aus dem Theorem folgt, dass die co-Probleme von HALTEPROBLEM, KORREKTHEIT, ERREICHBARER-CODE und ENTSCHEIDUNGSPROBLEM nicht semi-entscheidbar sind.

Jenseits der Semi-Entscheidbarkeit

- Bis jetzt haben wir nur Probleme $\mathcal{P}$ gesehen, für die zumindest eines der Probleme $\mathcal{P}$ und co-$\mathcal{P}$ semi-entscheidbar ist.
- Frage: Gibt es Probleme, sodass weder $\mathcal{P}$ noch co-$\mathcal{P}$ semi-entscheidbar ist?
- Antwort: Ja, solche Probleme gibt es. Wir werden 3 Beispiele von Problemen sehen, bei denen man sowohl für positive als auch für negative Instanzen implizit das co-Halteproblem lösen müsste.

Beispiele für Nicht-Semi-Entscheidbarkeit (1)

Selber-Output

Instanz: Programme Π_1, Π_2 und Input String I.
Frage: Haben Π_1 und Π_2 auf Input I das gleiche Verhalten?
Das heißt: entweder terminieren Π_1 und Π_2 bei Input I und liefern das gleiche Ergebnis oder beide Programme terminieren nicht bei Input I?

Intuition:

- Wir können positive Instanzen, bei denen keines der Programme Π_1 und Π_2 auf Input I terminiert, nicht erkennen. Denn wenn wir das könnten, wäre Nicht-Termination (sprich: das co-Halteproblem) semi-entscheidbar. Ein Widerspruch!
- Wir können aber auch nicht negative Instanzen erkennen, bei denen eines der Programme Π_1 und Π_2 auf I terminiert und das andere nicht. Denn wenn wir das könnten, würde das wiederum bedeuten, dass Nicht-Termination (sprich: das co-Halteproblem) semi-entscheidbar wäre. Ein Widerspruch!

Beispiele für Nicht-Semi-Entscheidbarkeit (2)

Programm-Äquivalenz

Instanz: 2 Programme Π_1, Π_2 (die beide als Input einen String über einem vorgegebenen Alphabet Σ nehmen).

Frage: Sind Π_1 and Π_2 äquivalent?

D.h.: gilt für jeden Input $I \in \Sigma^*$, dass entweder Π_1 und Π_2 terminieren und das gleiche Ergebnis liefern oder beide Programme endlos laufen?

Intuition:

- Das Problem Selber-Output vergleicht das Verhalten von 2 Programmen auf *einem* Input I, wohingegen das Problem Programm-Äquivalenz das Verhalten von 2 Programmen auf *uendlich vielen* Inputs I vergleicht.
- Wenn also schon das einfachere Problem und sein co-Problem nicht semi-entscheidbar sind, dann auch das allgemeinere Problem.

Beispiele für Nicht-Semi-Entscheidbarkeit (3)

UNIVERSELLES-HALTEPROBLEM

INSTANZ: Programm Π (das als Input einen String über einem vorgegebenen Alphabet Σ nimmt).

FRAGE: Hält Π auf jedem beliebigen Input String $I \in \Sigma^*$?

Intuition:

- Wir können positive Instanzen nicht erkennen, weil wir für unendlich viele Instanzen $I \in \Sigma^*$ überprüfen müssten, ob Π auf Input I hält.
- Wir können negative Instanzen nicht erkennen, weil wir dafür die Nicht-Termination von Π auf irgendeinem String I erkennen müssten. Das entspricht aber dem co-HALTEPROBLEM.

Bemerkung: Wir haben für die Nicht-Semi-Entscheidbarkeit bis jetzt nur die Intuition beschrieben. Für einen formalen Beweis benötigen wir den Begriff der Reduktion (= das wichtigste Werkzeug, um die Schwierigkeit beim Lösen von 2 Problemen zu vergleichen!).

Teil 1: Einführung in Berechenbarkeit und Entscheidbarkeit

Teil 1.1: Probleme, Programme, Algorithmen

Teil 1.2: Berechenbarkeit, Entscheidbarkeit

Teil 1.3: Semi-Entscheidbarkeit

Teil 1.4: Komplement

Teil 1.5: Jenseits der Semi-Entscheidbarkeit

Teil 1.6: Reduktionen

Erste Idee von Reduktionen

Idee (nicht auf die Theoretische Informatik beschränkt)

- Angenommen, wir wollen ein neues Problem A lösen, (d.h.: wir wollen einen Algorithmus für das Problem A entwickeln).
- Und angenommen, wir wissen, wie man ein verwandtes Problem B löst (d.h.: wir haben bereits einen passenden Algorithmus für das Problem B).
- Idee. Wir könnten versuchen, A zu lösen, indem wir es zum Problem B umformen, (d.h.: wir entwickeln einen Algorithmus für Problem A, der den Algorithmus für Problem B verwendet).

Schlussfolgerung: Wenn diese Strategie funktioniert, sagen wir:
"Problem A wird auf Problem B reduziert" und wir schreiben $A \leq B$.
$\Longrightarrow$ Problem A ist mindestens so leicht lösbar ist wie Problem B.

Zweite Idee von Reduktionen

Idee (typisch für Berechenbarkeit und Komplexität)

- Angenommen, es gelingt uns nicht, für ein neues Problem B ein geeigtnes Lösungsverfahren zu finden, d.h.: wir finden keinen (effizienten) Algorithmus oder nicht einmal ein Semi-Entscheidungsverfahren für Problem B.
- Und angenommen, wir wissen, dass ein verwandtes Problem A *schwer zu lösen* ist, d.h.: es wurde bereits bewiesen, dass es keinen (effizienten) Algorithmus oder nicht einmal ein Semi-Entscheidungsverfahren für Problem A gibt.
- Idee. Wir entwickeln einen Algorithmus für Problem A, der einen (hypothetischen) Algorithmus für Problem B verwendet.

Schlussfolgerung: Wenn diese Strategie funktioniert, sagen wir wiederum: "Problem A wird auf Problem B reduziert" und wir schreiben $A \leq B$.
$\Longrightarrow$ Problem B ist mindestens so schwer lösbar ist wie Problem A.

Beschränkung der Ressourcen

Motivation

- Eine Problemreduktion von A nach B sollte einfacher sein als die beiden betrachteten Probleme.
- Wenn das nicht der Fall wäre, könnte die Reduktion einen Teil der Komplexität von A beseitigen und ein Vergleich zwischen (den Schwierigkeitsgraden von) A und B wäre nicht mehr möglich.

Typische Anforderung in der Berechenbarkeitstheorie

Reduktionen müssen berechenbar sein.

Typische Anfordering in der Komplexitätstheorie

Reduktionen müssen effizient berechenbar sein, z.B. in polynomieller Zeit, d.h.: $O(n^k)$ für Instanzen der Größe n und Konstante $k \geq 1$.

Erste Art von Reduktionen: “Turing” Reduktionen

Idee

- Wir konstruieren ein Lösungsverfahren für Problem A, das ein Lösungsverfahren für Problem B als Unterprozedur verwendet.
- Das Lösungsverfahren für Problem A besteht also aus einem Steuerungsprogramm, das die Unterprozedur für Instanzen von Problem B “beliebig” oft aufrufen darf.
- Ressourcenbeschränkung: Das Steuerungsprogramm selbst muss ein (effizienter) Algorithmus sein. Im Fall einer Beschränkung auf polynomielle Zeit, heißt so eine Reduktion “Cook Reduktion”.

Zweite Art von Reduktionen: “Many-One” Reduktionen

Idee

- Definiere eine Funktion R von der Menge der Instanzen von Problem A auf die Menge der Instanzen von Problem B, d.h.: jede Instanz x von Problem A wird auf eine Instanz $R(x)$ von Problem B abgebildet.
- Wenn man nun die Instanz $R(x)$ mit einem Lösungsverfahren für Problem B löst, dann ist das Ergebnis für die Instanz $R(x)$ bereits das korrekte Ergebnis für die Instanz x des Problems A.
- In diesem Fall sagen wir: x und $R(x)$ sind äquivalent.
  Für Entscheidungsprobleme A und B bedeutet das: x ist eine positive Instanz von $A \Leftrightarrow R(x)$ ist eine positive Instanz von B.
- Ressourcenbeschränkung: Die Funktion R muss (effizient) berechenbar sein. Im Fall einer Beschränkung auf polynomielle Zeit, heißt so eine Reduktion “Karp Reduktion”.
- Bemerkung: Many-One Reduktionen sind ein Spezialfall von Turing Reduktionen: die Unterprozedur für Problem B wird exakt einmal aufgerufen und die Berechnung endet unmittelbar danach.

Einige klassische Entscheidungsprobleme der Aussagenlogik

SAT (Satisfiability)

Instanz: Aussagenlogische Formel ϕ.

Frage: Ist ϕ erfüllbar? (d.h.: gibt es eine Wahrheitsbelegung I für die aussagenlogischen Variablen in ϕ, sodass ϕ von I erfüllt wird).

3-SAT

Instanz: Aussagenlogische Formel ϕ in 3-CNF (d.h.: konjunktive Normalform, bei der jede Klausel aus höchstens 3 Literalen besteht).

Frage: Ist ϕ erfüllbar?

2-SAT

Instanz: Aussagenlogische Formel ϕ in 2-CNF (d.h.: konjunktive Normalform, bei der jede Klausel aus höchstens 2 Literalen besteht).

Frage: Ist ϕ erfüllbar?

Reduktion von 2-SAT auf GRAPH-ERREICHBARKEIT

Theorem

Das 2-SAT Problem lässt sich mittels Turing Reduktion auf das GRAPH-ERREICHBARKEIT Problem reduzieren.

Bemerkung:

- Das GRAPH-ERREICHBARKEIT Problem ist ein bekanntes, effizient lösbares Problem (sogar in linearer Zeit).
- Wir haben hier also den Fall $A \leq B$, bei dem es für Problem B ($=$ GRAPH-ERREICHBARKEIT) ein effizientes Lösungsverfahren gibt.
- Die Reduktion, die wir hier betrachten, lässt sich jedenfalls in polynomieller Zeit realisieren.
- Schlussfolgerung: Wir bekommen damit ein Lösungsverfahren für Problem A ($=$ 2-SAT), das in polynomieller Zeit läuft.

Reduktion von 2-SAT auf Graph-Erreichbarkeit

Notation: Für ein Literal α (d.h., aussagenlogische Variable x_i oder ihre Verneinung $\neg x_i$) bezeichnet $\overline{\alpha}$ das *duale* Literal, d.h.: wenn $\alpha = x_i$ ist, dann ist $\overline{\alpha} = \neg x_i$; und wenn $\alpha = \neg x_i$ ist, dann ist $\overline{\alpha} = x_i$.

Von der Formel zum Graphen

Sei ϕ eine beliebige Instanz von 2-SAT mit Variablen $X = \{x_1, \ldots, x_n\}$. Wir definieren daraus folgenden Graphen $G_\phi = (V, E)$:

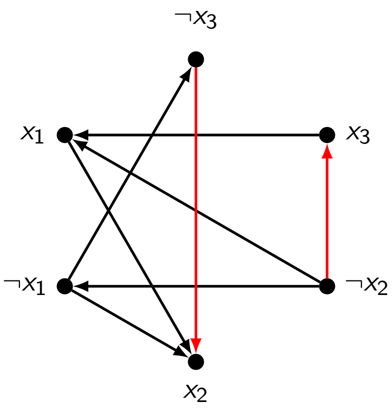
- Wir setzen $V = \{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$, d.h.:
  Die Knotenmenge V von G_ϕ enthält die Literale, die man mit den aussagenlogischen Variablen in X bilden kann.
- Die Kantenmenge E von G_ϕ enthält alle Kanten der Form $(\overline{\alpha}, \beta)$ und $(\overline{\beta}, \alpha)$, für die die Formel ϕ eine Klausel $\alpha \vee \beta$ enthält.

Intuition dieser Konstruktion: Eine Klausel der Form $\alpha \vee \beta$ ist logisch äquivalent zu den Implikationen $\overline{\alpha} \to \beta$ und $\overline{\beta} \to \alpha$. Auf diese Weise wollen wir logische Implikation mittels Pfaden im Graph codieren.

Example

$$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$$

$$\begin{aligned} \neg x_2 &\rightarrow x_3 \\ \neg x_3 &\rightarrow x_2 \end{aligned}$$



Zusammenhang zwischen ϕ und G_ϕ

Beobachtung (ohne Beweis)

Seien α, β zwei beliebige Literale über den Variablen X in ϕ (und somit auch zwei Knoten in G_ϕ). Dann sind folgende Behauptungen äquivalent:

(1) In jedem Modell I von ϕ ist $\alpha \rightarrow \beta$ true. (geschrieben: $\phi \models (\alpha \rightarrow \beta)$)

(2) Im Graph G_ϕ gibt es einen Pfad von α nach β.

Problematische Pfade im Graph G_ϕ

- Frage: Kann ϕ ein Modell haben, wenn es in G_ϕ einen Pfad von x_i nach $\neg x_i$ gibt? (d.h.: es gilt $\phi \models (x_i \rightarrow \neg x_i)$?)
  Antwort: Ja. Aber in jedem Modell von ϕ muss $I(x_i) =$ false gelten.
- Frage: Kann ϕ ein Modell haben, wenn es in G_ϕ einen Pfad von $\neg x_i$ nach x_i gibt? (d.h.: es gilt $\phi \models (\neg x_i \rightarrow x_i)$?)
  Antwort: Ja. Aber in jedem Modell von ϕ muss $I(x_i) =$ true gelten.
- Frage: Und wenn es sowohl einen Pfad von x_i nach $\neg x_i$ als auch von $\neg x_i$ nach x_i in G_ϕ gibt?

Beobachtung (ohne Beweis)

Folgende Behauptungen sind äquivalent:

(1) Die Formel ϕ ist erfüllbar.

(2) Es gibt keine Variable $x_i \in X$, für die der Graph G_ϕ sowohl einen Pfad von x_i nach $\neg x_i$ als auch von $\neg x_i$ nach x_i enthält.

Beweisidee: Die Richtung (1) $\Rightarrow$ (2) ist klar nach den Überlegungen auf der vorigen Folie. Die Richtung (2) $\Rightarrow$ (1) ist nicht-trivial. Man kann letztlich ein Modell I von ϕ in mehreren Stufen konstruieren:

- Zuerst werden in I alle Variablen x_i mit false bzw. true belegt, für die es einen Pfad von x_i nach $\neg x_i$ bzw. von $\neg x_i$ nach x_i in G_ϕ.
- Dann werden Wahrheitswerte entlang von allen Pfaden propagiert, z.B.: wenn $I(\alpha) = \text{true}$ gilt und es einen Pfad von α nach β gibt, dann muss auch $I(\beta) = \text{true}$ gelten.
- Wenn dann noch Variablen in I unbestimmt sind, kann man für *eine* dieser Variablen einen beliebigen Wahrheitswert festlegen und diesen dann wieder entlang von Pfaden propagieren.

Damit erhalten wir folgende Turing Reduktion von 2-SAT nach GRAPH-ERREICHBARKEIT:

procedure test2SAT:
Input: propositional formula ϕ in 2-CNF;
Output: true if ϕ is satisfiable and false otherwise.

from ϕ construct G_ϕ;
for all variables x in ϕ **do** {
  if G_ϕ contains a path from x to $\neg x$
    and a path from $\neg x$ to x **then return** false;}
return true;

Bemerkung:

- Beim Test in der if-Anweisung wird jeweils zwei Mal ein Lösungsverfahren für GRAPH-ERREICHBARKEIT als Unterprozedur aufgerufen, und zwar mit den Instanzen $(G_\phi, x, \neg x)$ und $(G_\phi, \neg x, x)$.
- Korrektheit der Reduktion = Korrektheit der Prozedur test2SAT. (diese folgt aus den vorigen Beobachtungen und Überlegungen).

Ein klassisches Entscheidungsproblem der Graphentheorie

k-FÄRBBARKEIT

Sei $k \in \mathbb{N}$ mit $k \geq 2$. Dann ist k-FÄRBBARKEIT folgendermaßen definiert:

INSTANZ: ungerichteter Graph $G = (V, E)$
FRAGE: Ist der Graph G k-färbbar?
d.h.: gibt es eine Funktion $f : V \to \{0, \dots, k-1\}$, so dass für alle Kanten $[v_i, v_j] \in E$ gilt: $f(v_i) \neq f(v_j)$.

Bemerkung:

- Man kann k-Färbbarkeit für beliebige Werte $k \geq 2$ betrachten.
- Im Komplexitätsteil der Vorlesung wird gezeigt werden, dass 3-Färbbarkeit (eig. k-Färbbarkeit für jedes $k \geq 3$) ein schweres Problem ist (d.h. es ist kein effizienter Algorithmus bekannt)
- Wir betrachten hier 2-Färbbarkeit (ein sehr leichtes Problem).

Reduktion von 2-Färbbarkeit auf 2-SAT

Theorem

Das 2-Färbbarkeit Problem lässt sich mittels Many-One Reduktion auf das 2-SAT Problem reduzieren.

Probemreduktion

Sei $G = (V, E)$ eine beliebige Instanz von 2-Färbbarkeit, d.h.:
$G = (V, E)$ ist ein ungerichteter Graph mit Knotenmenge $V = \{v_1, \ldots, v_n\}$ und Kantenmenge E.

Daraus definieren wir folgende Instanz φ_G von 2-SAT mit den aussagenlogischen Variablen $x_1, \ldots, x_n$:

$$\varphi_G = \bigwedge_{[v_i, v_j] \in E} \Big((x_i \vee x_j) \wedge (\neg x_i \vee \neg x_j) \Big)$$

Bemerkung:

- Wir haben gerade gezeigt, dass das 2-SAT Problem effizient lösbar ist.
- Aufgrund der Reduktion von 2-Färbbarkeit auf 2-SAT ist natürlich auch 2-Färbbarkeit effizient lösbar (wobei das im Fall von 2-Färbbarkeit auch ohne diese Reduktion leicht zu sehen wäre).
- Reduzierbarkeit ist transitiv: Man könnte nun die Reduktion von 2-Färbbarkeit auf 2-SAT und die Reduktion von 2-SAT auf Graph-Erreichbarkeit hintereinander ausführen, und würde damit eine Reduktion von 2-Färbbarkeit auf Graph-Erreichbarkeit bekommen.

Korrektheit der Reduktion

Wir müssen folgende Äquivalenz zeigen:
$G = (V, E)$ ist eine positive Instanz von 2-FÄRBBARKEIT $\Leftrightarrow$
φ_G ist eine positive Instanz von 2-SAT.

Üblicherweise geht man so vor, dass man die beiden Implikationen "$\Rightarrow$" und "$\Leftarrow$" getrennt zeigt.

Wir zeigen hier die Implikation "$\Rightarrow$", d.h.:
$G = (V, E)$ ist eine positive Instanz von 2-FÄRBBARKEIT $\Rightarrow$
φ_G ist eine positive Instanz von 2-SAT.

Die andere Richtung ist Teil des ersten Übungsblatts.

Korrektheit der Reduktion: Beweis der “$\Rightarrow$”-Richtung

Angenommen $G = (V, E)$ ist eine positive Instanz von 2-FÄRBBARKEIT. Dann gibt es eine Funktion $f : V \to \{0, 1\}$, die eine gültige 2-Färbung für den Graph $G = (V, E)$ ist, d.h.: für jede Kante $[v_i, v_j]$ in G gilt entweder $(f(v_i) = 0$ und $f(v_j) = 1)$ oder $(f(v_i) = 1$ und $f(v_j) = 0)$.

Wir definieren daraus folgende Wahrheitsbelegung I auf den aussagenlogischen Variablen $x_1, \ldots, x_n$:

$$I(x_i) = \begin{cases} \text{false} & \text{falls } f(v_i) = 0 \\ \text{true} & \text{falls } f(v_i) = 1 \end{cases}$$

Wir müssen noch zeigen, dass I die Formel ϕ_G erfüllt. Die Formel ϕ_G ist eine Konjunktion von Teilformeln der Form $(x_i \vee x_j) \wedge (\neg x_i \vee \neg x_j)$. Es reicht daher zu zeigen, dass I jede Klausel $(x_i \vee x_j)$ und $(\neg x_i \vee \neg x_j)$ erfüllt.

Beweis, dass I jede Klausel $(x_i \vee x_j)$ und $(\neg x_i \vee \neg x_j)$ erfüllt:

(1) Beliebige Klausel der Form $(x_i \vee x_j)$. Laut Problemreduktion enthält ϕ_G so eine Klausel nur dann, wenn es eine Kante $[v_i, v_j]$ in G gibt.

   Laut Annahme ist f eine gültige Zweifärbung von G. Also ist mindestens einer der Funktionswerte $f(v_i)$ und $f(v_j)$ gleich 1.

   Laut Definition von I gilt dann für mindestens eine der Variablen x_i und x_j, dass sie in I den Wahrheitswert true hat. Somit erfüllt I die Klausel $(x_i \vee x_j)$.

(2) Beliebige Klausel der Form $(\neg x_i \vee \neg x_j)$. Laut Problemreduktion enthält ϕ_G so eine Klausel nur dann, wenn es eine Kante $[v_i, v_j]$ in G gibt. Laut Annahme ist f eine gültige Zweifärbung von G. Also ist mindestens einer der Funktionswerte $f(v_i)$ und $f(v_j)$ gleich 0.

   Laut Definition von I gilt dann für mindestens eine der Variablen x_i und x_j, dass sie in I den Wahrheitswert false hat. Dann hat aber mindestens eines der Literale $\neg x_i$ und $\neg x_j$ den Wahrheitswert true. Somit erfüllt I die Klausel $(\neg x_i \vee \neg x_j)$.

Typische Vorgangsweise bei Korrektheitsbeweisen

- Zur Erinnerung: eine Many-One Reduktion R von Problem A auf Problem B ist korrekt, wenn für jede beliebige Instanz x von A die Äquivalenz gilt: x ist eine positive Instanz von $A \Leftrightarrow R(x)$ ist eine positive Instanz von B.
- Üblicherweise zeigt man die 2 Richtungen "$\Rightarrow$" und "$\Leftarrow$" der Äquivalenz getrennt.
- Der Beweis beginnt immer mit der Annahme "Sei x eine positive Instanz von A." bzw. "Sei $R(x)$ eine positive Instanz von B.".
- Mittels *schlüssiger* Beweisschritte versucht man zu zeigen, dass man dann auch beim anderen Problem eine positive Instanz hat.
- Typische Beweisschritte sind, wenn etwas *aufgrund der Annahme*, *aufgrund einer Definition* (z.B. einer "Lösung") oder *aufgrund der Problemreduktion* gilt.
- Es erhöht die Lesbarkeit eines Beweises, wenn man im Beweis klarstellt, was man gerade möchte bzw. was noch zu zeigen ist.

Unentscheidbarkeitsbeweise mittels Reduktionen

- Wir wissen, dass das HALTEPROBLEM unentscheidbar ist.
- Außerdem haben wir gezeigt, dass es noch viele weitere (sogar überabzählbar viele) unentscheidbare Probleme gibt.
- Um die Unentscheidbarkeit von einem weiteren Problem $\mathcal{P}$ zu beweisen, könnten wir einen ähnlich komplizierten Beweis wie das Diagonalargument für das HALTEPROBLEM suchen.
- Bessere Idee. Um die Unentscheidbarkeit von einem (neuen) Problem $\mathcal{P}$ zu beweisen, können wir das HALTEPROBLEM (oder irgendein anderes, als untenscheidbar bewiesenes Problem) auf $\mathcal{P}$ reduzieren.
- Begründung: Angenommen es gäbe eine Entscheidungsprozedur $\Pi_{\mathcal{P}}$ für das Problem $\mathcal{P}$. Dann konstruieren wir eine Entscheidungsprozedur Π_h für das HALTEPROBLEM wie folgt:
  - Π_h nimmt als Input eine Instanz $x = (\Pi, I)$ des HALTEPROBLEMS;
  - Π_h wendet die Problemreduktion R vom HALTEPROBLEM auf $\mathcal{P}$ auf diese Instanz x an, d.h.: es wird $R(x)$ berechnet;
  - mit $R(x)$ als Input wird $\Pi_{\mathcal{P}}$ aufgerufen:
  - der Output von $\Pi_{\mathcal{P}}$ ist auch der Output von Π_h.

Beispiele von unentscheidbaren Problemen

Wiederholung: KORREKTHEIT

INSTANZ: (Quellcode von einem) Programm Π und 2 Strings I_1, I_2.
FRAGE: Liefert Π bei Aufruf mit Input I_1 als Output den String I_2?

Theorem

Das KORREKTHEIT *Problem ist unentscheidbar.*

Beweis mittels Reduktion vom HALTEPROBLEM

Sei (Π, I) eine beliebige Instanz des HALTEPROBLEMS, d.h.: Π ist ein SIMPLE Programm und I ist ein möglicher Input-String für Π.

Daraus definieren wir eine Instanz (Π', I_1, I_2) des KORREKTHEIT Problems, indem wir $I_1 = I_2 = I$ setzen und Π' wie folgt definieren:

```
String Π′ (String S) {
    call Π(S);
    return S; }
```

Fortsetzung des Unentscheidbarkeitsbeweises

Wir müssen noch die Korrektheit der Reduktion zeigen, d.h.: wir müssen zeigen, dass die ursprüngliche Instanz des HALTEPROBLEMS und die (in der Problemreduktion) definierte Instanz des KORREKTHEIT Problems äquivalent sind, d.h.:

(Π, I) ist eine positive Instanz des HALTEPROBLEMS $\Leftrightarrow$
(Π', I_1, I_2) ist eine positive Instanz des KORREKTHEIT Problems.

"$\Rightarrow$" Angenommen (Π, I) ist eine positive Instanz des HALTEPROBLEMS, d.h.: Π hält bei Input I. Laut Konstruktion von Π', hält dann auch Π' auf dem Intput $I_1 = I$ und liefert als Output den String $I_2 = I$, d.h.: (Π', I_1, I_2) ist eine positive Instanz des KORREKTHEIT Problems.

"$\Leftarrow$" Angenommen (Π', I_1, I_2) ist eine positive Instanz des KORREKTHEIT Problems, d.h.: Π' hält bei Input I_1 und liefert I_2 als Output. Laut Problemreduktion gilt $I_1 = I_2 = I$ und der Aufruf von Π' mit Input I führt zu einem Aufruf von Π mit Input I. Da Π' auf diesem Input hält (und I_2 liefert), muss auch Π auf dem Input I halten, d.h.: (Π, I) ist eine positive Instanz des HALTEPROBLEMS.

Beispiele von unentscheidbaren Problemen (Forts.)

Wiederholung: SELBER-OUTPUT

INSTANZ: Programme Π_1, Π_2 und Input String I.
FRAGE: Haben Π_1 und Π_2 auf Input I das gleiche Verhalten?
Das heißt: entweder terminieren Π_1 und Π_2 bei Input I und liefern das gleiche Ergebnis oder beide Programme terminieren nicht bei Input I?

Theorem

Das SELBER-OUTPUT *Problem ist unentscheidbar.*

Beweis mittels Reduktion vom HALTEPROBLEM

Sei (Π, I) eine beliebige Instanz des HALTEPROBLEMS, d.h.: Π ist ein SIMPLE Programm und I ist ein möglicher Input-String für Π.

Daraus definieren wir eine Instanz (Π_1, Π_2, I') des SELBER-OUTPUT Problems mit $I' = I$ und folgenden Programmen Π_1, Π_2:

```
String Π1 (String S) {
return S; }
```

```
String Π2 (String S) {
   call Π(S);  return S; }
```

Intuition der Reduktion:

- Programm Π_1 liefert bei Input I *unmittelbar* den Output I. Offensichtlich hält Programm Π_1 garantiert.
- Programm Π_2 ruft zunächst Π mit Input I auf und gibt dann den Output I aus – vorausgesetzt, dass Π bei Input I hält.

Fortsetzung des Unentscheidbarkeitsbeweises

Wir müssen noch die Korrektheit der Reduktion zeigen, d.h.:

(Π, I) ist eine positive Instanz des HALTEPROBLEMS $\Leftrightarrow$
(Π_1, Π_2, I) ist eine positive Instanz des KORREKTHEIT Problems.

Wir zeigen sofort beide Richtungen der Äquivalenz:

(Π, I) ist positive Instanz des HALTEPROBLEMS, d.h.: Π hält bei Input I.
$\Leftrightarrow$ Programm Π_2 terminiert bei Input I und liefert den Output I.
$\Leftrightarrow$ Π_1 und Π_2 haben auf Input I das gleiche Verhalten. (Da Programm Π_1 immer terminiert und bei Input I den Output I liefert.)

Beweise der Nicht-Semi-Entscheidbarkeit

Idee:

- Wir wissen, dass das co-HALTEPROBLEM nicht semi-entscheidbar ist.
- Wir können daher die Nicht-Semi-Entscheidbarkeit eines (neuen) Problems mittels Reduktion vom co-HALTEPROBLEM beweisen.

Beobachtung:

- Angenommen wir können für 2 Probleme zeigen, dass $\mathcal{P}_1 \leq \mathcal{P}_2$ gilt. Dann gilt offensichtlich auch co-$\mathcal{P}_1 \leq$ co-$\mathcal{P}_2$.
- Begründung: Korrektheit einer Reduktion R bedeutet, dass für jede Instanz x von $\mathcal{P}_1$ gilt: x ist positive Instanz von $\mathcal{P}_1 \Leftrightarrow R(x)$ ist positive Instanz von $\mathcal{P}_2$. Dann gilt natürlich auch: x ist negative Instanz von $\mathcal{P}_1$ (und somit positive Instanz von co-$\mathcal{P}_1$) $\Leftrightarrow R(x)$ ist negative Instanz von $\mathcal{P}_2$ (und somit positive Instanz von co-$\mathcal{P}_2$).
- Daher gilt: $\mathcal{P}_1 \leq \mathcal{P}_2 \Leftrightarrow$ co-$\mathcal{P}_1 \leq$ co-$\mathcal{P}_2$
- Schlussfolgerung: Die Probleme co-KORREKTHEIT und co-SELBER-OUTPUT sind nicht semi-entscheidbar.

Weitere nicht-semi-entscheidbare Probleme

Wir haben gerade gesehen, dass das co-SELBER-OUTPUT Problem nicht semi-entscheidbar ist. Es gilt aber auch:

Theorem

Das SELBER-OUTPUT Problem ist nicht semi-entscheidbar.

Beweis mittels Reduktion vom co-HALTEPROBLEM

Sei (Π, I) eine beliebige Instanz des co-HALTEPROBLEMS, d.h.: Π ist ein SIMPLE Programm und I ist ein möglicher Input-String für Π.

Daraus definieren wir eine Instanz (Π_1, Π_2, I') des SELBER-OUTPUT Problems mit $I' = I$ und folgenden Programmen Π_1, Π_2:

```
String Π1 (String S) {
while (true) do {} };
```

```
String Π2 (String S) {
   call Π(S);  return S; }
```

Intuition der Reduktion:

- Programm Π_1 ignoriert den Input I und läuft sofort in eine *Endlosschleife*, d.h.: Π_1 terminiert garantiert nicht.
- Programm Π_2 ruft zunächst Π mit Input I auf und gibt dann den Output I aus, d.h.: Π_2 terminiert auf Input I genau dann, wenn Π auf Input I terminiert.

Fortsetzung des Beweises

Wir müssen noch die Korrektheit der Reduktion zeigen, d.h.:

(Π, I) ist eine positive Instanz des co-HALTEPROBLEMS $\Leftrightarrow$
(Π_1, Π_2, I) ist eine positive Instanz des SELBER-OUTPUT Problems.

Wir zeigen sofort beide Richtungen der Äquivalenz:

(Π, I) ist eine positive Instanz des co-HALTEPROBLEMS, d.h.:
Π hält bei Input I nicht. $\Leftrightarrow$
Programm Π_2 terminiert auf Input I nicht. $\Leftrightarrow$
Π_1 und Π_2 haben auf Input I das gleiche Verhalten.
(Da Programm Π_1 bei jedem Input endlos läuft.)

Zusammenfassung von Teil 1 der LVA

- Formale Definition von “Problemen” (Instanzen, Frage)
- verschiedene Arten von Problemen (insbes.: Entscheidungsprobleme)
- Modell für “Algorithmen” (SIMPLE Programmiersprache)
- Berechenbarkeit, Entscheidbarkeit, Semi-Entscheidbarkeit
- Abzählbarkeit, Überabzählbarkeit (Diagonalargument)
- HALTEPROBLEM (noch ein Diagonalargument)
- co-Probleme und ihre Eigenschaften: (Semi-)Entscheidbarkeit
- unentscheidbare / nicht-semi-entscheidbare Probleme (Intuition)
- Reduktionen (Motivation, Turing vs. Many-One)
- Korrektheitsbeweis von Reduktionen

192.017 Theoretische Informatik und Logik, VU 4.0
Teil 2a: Formale Sprachen

Marion OSWALD

Wintersemester 2023

Teil 2a: Formale Sprachen

Inhalt

- Grundlagen (Wiederholung)
- Turingmaschinen
- Reguläre Sprachen (Wiederholung)
- Grammatiken
- Kontextfreie Grammatiken und Sprachen
- Jenseits der Kontextfreiheit
- Chomsky Hierarchie

Grundlagen: Wörter

Alphabet: *endliche*, nicht-leere Menge atomarer Symbole (Σ, T)

Wort über Σ: endliche Folge von Symbolen aus Σ

Länge eines Wortes w über Σ (geschrieben $|w|$): Anzahl der Symbole, die w enthält

Leerwort: Wort mit der Länge 0, geschrieben ε, d.h. $|\varepsilon| = 0$

Für ein Wort w über Σ und ein Symbol $a \in \Sigma$ bezeichnen wir die **Anzahl der Symbole** a **in** w mit $|w|_a$.

Konkatenation: Hintereinanderschreiben von Wörtern
Seien x, y Wörter mit $|x| = n$, $|y| = m$, dann ist $x \cdot y = xy$ und $|xy| = n + m$

Potenzbildung: Verkettung eines Wortes w mit sich selbst, wobei $w^0 = \varepsilon$ und $w^n = ww^{n-1}$

Sei $w = a_1 a_2 \dots a_{n-1} a_n$, dann ist $w^r = a_n a_{n-1} \dots a_2 a_1$ das **Spiegelbild** von w.

Ein Wort w heißt **Palindrom**, wenn $w = w^r$ gilt.

Grundlagen: (Formale) Sprachen

Σ^+: Menge aller (nicht-leeren) Wörter über Σ
Σ^*: Menge aller Wörter (inklusive ε) über Σ

Formale Sprache: beliebige Teilmenge L von Σ^*, $L \subseteq \Sigma^*$

$(\Sigma^*, \cdot, \varepsilon)$ bildet ein *Monoid*.

Beispiele für Sprachen

- $\{\}$
- $\{\varepsilon\}$
- $\{\varepsilon, ab, abab, ababab, ...\} = \{(ab)^n \mid n \geq 0\} \quad \Sigma = \{a, b\}$
- $\{\varepsilon, ab, aabb, aaabbb, ...\} = \{a^n b^n \mid n \geq 0\} \quad \Sigma = \{a, b\}$
- $\{w \in \Sigma^* \mid w = w^r\} \quad \Sigma = \{0, 1\}$
- $\{w \in \{a, b, c\}^* \mid |w|_a = |w|_b = |w|_c\}$

$\mathcal{P}(\Sigma^*)$: **Menge aller Sprachen** $L \subseteq \Sigma^*$

Operationen auf Sprachen

Konkatenation: $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}$

Bsp: $\{ab, b\}\{a, bb\} = \{aba, abbb, ba, bbb\}$
(Achtung: $\{ab, b\} \times \{a, bb\} = \{(ab, a), (ab, bb), (b, a), (b, bb)\}$)

Potenzbildung: $L^n = \{w_1 ... w_n \mid w_1, ..., w_n \in L\}$
($L^0 = \{\varepsilon\}$ und $L^{n+1} = LL^n$ für $n \geq 0$)

Bsp: $\{ab, ba\}^2 = \{abab, abba, baab, baba\}$

Kleene-Stern[1]: $L^* = \bigcup_{n \geq 0} L^n$ ($L^+ = \bigcup_{n \geq 1} L^n$)

Bsp: $\{01\}^* = \{\varepsilon, 01, 0101, 010101, ...\} \neq \{0, 1\}^*$

$(\mathcal{P}(\Sigma^*), \cdot, \{\varepsilon\})$ bildet ein *Monoid*.

- $\varepsilon \in L^*$ gilt für alle Sprachen L (z.B.: $\{\}^* = \{\varepsilon\}$)
- $\varepsilon \in L^+$ gilt für eine Sprache L genau dann, wenn $\varepsilon \in L$

[1] benannt nach Stephen Cole Kleene (1909 - 1994)

Rechenregeln für Sprachoperatoren

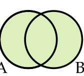
$$
\begin{aligned}
A \cdot (B \cdot C) &= (A \cdot B) \cdot C && \text{Assoziativität von} \cdot \\
A \cdot (B \cup C) &= A \cdot B \cup A \cdot C && \text{Distributivität von} \cdot \\
(B \cup C) \cdot A &= B \cdot A \cup C \cdot A && \text{Distributivität von} \cdot
\end{aligned}
$$

$$
\begin{aligned}
(A \cup \{\varepsilon\})^* &= A^* \\
(A^*)^* &= A^* \\
A \cdot A^* &= A^+ \\
A^* \cdot A &= A^+ \\
A^+ \cup \{\varepsilon\} &= A^*
\end{aligned}
\qquad
\begin{aligned}
\{\varepsilon\} \cdot A &= A \\
A \cdot \{\varepsilon\} &= A \\
\{\} \cdot A &= \{\} \\
A \cdot \{\} &= \{\}
\end{aligned}
$$

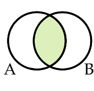
$(\mathcal{P}(\Sigma^*), \cup, \cdot, \{\}, \{\varepsilon\})$ bildet einen *nichtkommutativen Semiring*.

Mengenoperationen auf Sprachen

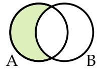
Vereinigung:
$A \cup B = \{x \in \Sigma^* \mid x \in A \text{ oder } x \in B\}$



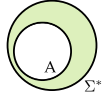
Durchschnitt:
$A \cap B = \{x \in \Sigma^* \mid x \in A \text{ und } x \in B\}$
(A und B sind **disjunkt** wenn $A \cap B = \{\}$)



Differenz:
$A - B = \{x \in \Sigma^* \mid x \in A \text{ und } x \notin B\}$



Komplement:
$\overline{A} = \Sigma^* - A = \{x \in \Sigma^* \mid x \notin A\}$



$A \subseteq B$: A ist eine **Teilmenge** von B (jedes Element von A ist auch ein Element von B)

$A \subset B$: A ist eine **echte Teilmenge** von B (A ist eine Teilmenge von B, die nicht gleich B ist)

$A = B$ wenn $A \subseteq B$ und $B \subseteq A$

Abgeschlossenheit

Sind eine Menge B und Operatoren auf Teilmengen von B gegeben, so kann man sich natürlich fragen, ob die Anwendung der Operatoren auf diese Teilmengen von B wiederum Teilmengen von B ergibt.

Sei B eine Menge und $f : B^n \longrightarrow B$ eine Funktion. Eine Menge $A \subseteq B$ heißt **abgeschlossen** unter f, wenn gilt:

$$x_1, \ldots, x_n \in A \Rightarrow f(x_1, \ldots, x_n) \in A$$

Beispiel

Die Menge der natürlichen Zahlen $\mathbb{N}$ ist abgeschlossen unter Addition, aber nicht unter Subtraktion.

Sprache - Problem

Mit den Begriffen “Sprache” und “Problem” ist eigentlich dasselbe gemeint. Beschäftigen wir uns nur mit Wörtern an sich, dann tendieren wir dazu, uns eine Menge von Wörtern als eine Sprache vorzustellen. Wird der durch das Wort repräsentierte Sachverhalt wichtiger als das Wort selbst, wird eine Menge von Wörtern eher als Problem verstanden.

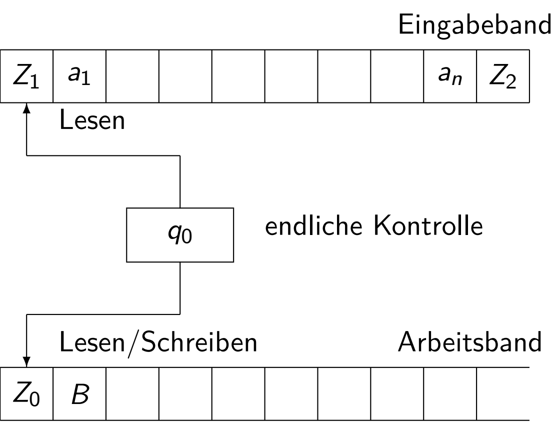
Beispiel: Problem vs. Sprache

Das Problem des Testens, ob eine Zahl eine Primzahl ist, kann durch die Sprache L_p ausgedrückt werden, die aus allen binären Zeichenreihen besteht, deren Wert als Binärzahl eine Primzahl darstellt:

$$L_p = \{bin(p) \mid p \text{ ist Primzahl } \}$$

D.h., ist ein aus Nullen und Einsen bestehendes Wort gegeben, gilt es zu entscheiden, ob die Zeichenreihe die Binärdarstellung einer Primzahl ist.

Turingmaschinen[2]



[2] Alan M. Turing (1912 - 1954)

Turingmaschinen

Eine Turingmaschine M besteht aus folgenden Komponenten:

- Das *Eingabeband* kann von links nach rechts gelesen werden. Es beinhaltet eine endliche Folge von Zeichen (das Eingabewort), wobei das Eingabewort vom Anfangssymbol Z_1 und vom Endsymbol Z_2 begrenzt ist.
- Das *Arbeitsband* kann beliebig gelesen und beschrieben werden.
- Die *endliche Kontrolle* kann einen Zustand aus einer endlichen Menge von Zuständen annehmen und steuert den Lesekopf auf dem Eingabeband und den Lese-/Schreibkopf auf dem Arbeitsband; die Kopfbewegungen sind L, R, S (links/left, rechts/right, stehenbleiben/stay).

Formale Beschreibung einer Turingmaschine

Eine *Turingmaschine* ist ein Achttupel

$$M = (Q, T, \Gamma, \delta, q_0, \{Z_0, Z_1, Z_2\}, B, F),$$

- Q endliche Menge von Zuständen,
- T Alphabet des Eingabebandes,
- Γ Alphabet des Arbeitsbandes,
- δ Übergangsfunktion,
- $q_0 \in Q$ Startzustand,
- $Z_0 \in \Gamma$ linkes Begrenzungssymbol auf Arbeitsband, Z_1, Z_2 Begrenzungssymbole des Eingabewortes,
- $B \in \Gamma$ Blanksymbol,
- $F \subseteq Q$ Menge von Endzuständen.

Übergänge

$$(q, a, X; p, Y, D_E, D_A) \in \delta$$

M liest im Zustand q auf dem Eingabeband Symbol a und auf Arbeitsband Symbol X

- wechselt nun davon abhängig in den Zustand p,
- überschreibt auf dem Arbeitsband Symbol X durch Symbol Y ,
- bewegt auf dem Eingabeband den Lesekopf in die durch D_E beschriebene Richtung und auf dem Arbeitsband den Lese-/Schreibkopf in die durch D_A beschriebene Richtung (D_E, $D_A \in \{L, R, S\}$).

(Deterministische) Turingmaschinen: akzeptierte Sprache

Akzeptierte Sprache

Die von der Turingmaschine M *akzeptierte Sprache* $L(M)$ besteht aus genau all jenen Wörtern, bei deren Anaylse M einen Endzustand erreicht.

Deterministische Turingmaschine

Eine Turingmaschine M heißt *deterministisch*, wenn für alle $(q, a, X) \in Q \times V_T \times \Gamma$ höchstens ein Element $(q, a, X; p, Y, D_E, D_A) \in \delta$ existiert; wir schreiben dann auch $\delta(q, a, X) = (p, Y, D_E, D_A)$.

Turingmaschinen - Beispiel

Turingmaschine, die $\{a^n b^n c^n \mid n \geq 1\}$ akzeptiert

$$M = (\{p, q, r, s\}, \{a, b, c\}, \{Z_0, A, B, C\}, \delta, p, \{Z_0, Z_1, Z_2\}, B, \{s\})$$

wobei

$$\begin{aligned}
1: &\quad \delta(p, a, B) = (p, A, R, R)\\
2: &\quad \delta(p, b, B) = (q, B, S, L)\\
3: &\quad \delta(q, b, A) = (q, C, R, L)\\
4: &\quad \delta(q, c, Z_0) = (r, Z_0, S, R)\\
5: &\quad \delta(r, c, C) = (r, B, R, R)\\
6: &\quad \delta(r, Z_2, B) = (s, B, S, L)
\end{aligned}$$

Turingmaschinen - Beispiel ctd.

Dabei passiert Folgendes:
1 : für jedes eingelesenes Symbol a wird ein A aufs Arbeitsband geschrieben;

2 : wenn das erste Symbol b eingelesen wird, geht M einen Schritt nach links; dieses erste Symbol b wird bei der ersten Anwendung von Übergang 3 nochmals gelesen;

3 : für ein eingelesenes Symbol b wird ein A auf dem Arbeitsband mit C überschrieben;

4 : wird das erste Symbol c gelesen, so muss M genau Z_0 erreicht haben;

5 : für jedes eingelesene Symbol c wird nun ein C auf dem Arbeitsband gelöscht (durch das Blanksymbol B ersetzt);

6 : wird mit dem Ende der Eingabe (d.h., dem Erreichen von Z_2) gleichzeitig das Ende der Symbole C auf dem Arbeitsband erreicht, so hält die Turingmaschine M im Endzustand s, und hat damit die Eingabe akzeptiert.

Normalform

Eine (deterministische) Turingmaschine ist in **Normalform**, wenn

- sie nur einen Endzustand besitzt,
- das Arbeitsband am Ende eines akzeptierenden Laufes der Turingmaschine leer ist und
- der letzte Übergang von der Gestalt $\delta(s, Z_2, Z_0) = (f, Z_0, S, R)$ ist, wobei f dieser einzige Endzustand und s ein beliebiger anderer Zustand ist.

Normalform: Beispiel

M' akzeptiert $\{a^n b^n c^n \mid n \geq 1\}$ in Normalform

$M' = (\{p, q, r, s, f\}, \{a, b, c\}, \{Z_0, A, B, C\}, \delta, p, \{Z_0, Z_1, Z_2\}, B, \{f\})$ wobei

Übergänge 1 bis 6 wie bei M aus vorigem Beispiel.

Damit M' jedoch mit leerem Band akzeptieren kann, benötigen wir zusätzlich noch folgende Übergänge:

$7 : \delta(s, Z_2, B) = (s, B, S, L)$ sowie
$8 : \delta(s, Z_2, Z_0) = (f, Z_0, S, R)$.

TM: eingeschränkte Varianten - LBA

Ein **linear beschränkter Automat** (LBA) ist eine Turingmaschine, die auf dem Arbeitsband nur höchstens soviel Platz verwendet wie das Eingabewort lang ist (der Platz darf sogar eine lineare Funktion der Länge des Eingabewortes sein).

LBA

Eine Turingmaschine M heißt *linear beschränkter Automat*, wenn es eine lineare Funktion $cn + d$ so gibt, dass die Turingmaschine M für jedes Eingabewort w der Länge n während der Analyse höchstens $cn + d$ Felder auf dem Arbeitsband benötigt.

TM: Eingeschränkte Varianten - Kellerautomaten

Turingmaschine erfüllt **Kellerautomatenbedingung**, wenn sie

- auf dem Eingabeband nie nach links gehen kann und
- für den Lese-/Schreibkopf auf dem Arbeitsband in jeder Konfiguration gilt, dass
  - links davon nur Nicht-Blanksymbole und
  - rechts davon nur Blanksymbole stehen können.

Kellerautomatenbedingung

Im Wesentlichen ist also ein Kellerautomat eine Turingmaschine, die das Arbeitsband als sogenannten Keller verwendet, d.h., der Kellerautomat darf bei jedem Schritt nur das oberste Symbol lesen und dieses dann löschen bzw. durch ein Symbol über dem sogenannten Kelleralphabet ersetzen.

Das linke Begrenzungssymbol des Arbeitsbandes Z_0 heißt dann *Kellergrundsymbol*.

Ein Kellerautomat funktioniert wie ein Stack – nach dem LIFO – Last-in-first-out-Prinzip.

Kellerautomaten: akzeptierte Sprache

Es gibt zwei Möglichkeiten, die vom Kellerautomaten akzeptierte Sprache zu definieren:

1. Kellerautomat befindet sich im Sinne einer Turingmaschine nach dem Einlesen des gesamten Eingabewortes in einem akzeptierenden Zustand.
2. Kellerautomat akzeptiert durch leeren Keller.

Normalform: Eingabewort wird durch Endzustand und leeren Keller akzeptiert d.h., beim Übergang in den (einzigen) Endzustand wird das rechte Begrenzungssymbol Z_2 des Eingabebandes und im Keller gleichzeitig das Kellergrundsymbol gelesen. (entspricht der Normalform für Turingmaschinen)

Kellerautomaten: Beispiel

Kellerautomat M in Normalform an, der $L = \{0^n 1^{2n+1} \mid n \geq 0\}$ akzeptiert

$$M = (\{q_0, q_1, q_2, q_f\}, \{0, 1\}, \{Z_0, A\}, \delta, q_0, \{Z_0, Z_1, Z_2\}, \{q_f\}),$$

wobei

$$\begin{aligned}
1: &\quad \delta(q_0, 0, B) = \{(q_1, A, S, R)\} \\
2: &\quad \delta(q_1, 0, B) = \{(q_0, A, R, R)\} \\
3: &\quad \delta(q_0, 1, B) = \{(q_2, B, R, L)\} \\
4: &\quad \delta(q_2, 1, A) = \{(q_2, B, R, L)\} \\
5: &\quad \delta(q_2, Z_2, Z_0) = \{(q_f, Z_0, S, R)\}
\end{aligned}$$

Kellerautomaten: Beispiel ctd.

Dabei passiert Folgendes:
$1, 2$: für jedes eingelesenes Symbol 0 werden 2 Symbole A in den Keller geschrieben;
3 : das erste Symbol 1 wird überlesen;
4: nun wird für jedes eingelesene Symbol 1 ein Symbol A im Keller gelöscht;
5 : wird gleichzeitig mit dem Ende der Eingabe (dem Erreichen von Z_2) das Kellergrundsymbol Z_0 erreicht, geht der Kellerautomat in den Enzustand q_f über und akzeptiert somit die Eingabe.

Turingmaschinen: Eingeschränkte Varianten - EA

Endlicher Automat: Turingmaschine, die das Arbeitsband gar nicht benötigt. Übergänge besitzen einfache Form $(q, a; p, D_E)$

Erlaubt man nur $D_E \in \{R, S\}$, kann die Übergangsfunktion δ durch Tripel der Gestalt $(q, a; p)$ beschreiben werden ($a \in T \cup \{\varepsilon\}$); dabei ist

$(q, a; p)$ für $a \in T$ als $(q, a; p, R)$ zu interpretieren und
$(q, \varepsilon; p)$ als $(q, a; p, S)$ für ein beliebiges $a \in T$.

Außerdem geht man davon aus, dass ein endlicher Automat seine Analyse auf dem ersten Symbol des Eingabewortes beginnt und in einem Endzustand akzeptiert, wenn der Lesekopf auf dem rechten Begrenzungssymbol steht.

Akzeptierte Sprachen

Eine formale Sprache heißt

0 **rekursiv aufzählbar**[3], wenn Sie von einer *Turingmaschine* akzeptiert wird

1 **kontextsensitiv**, wenn sie von einem *linear beschränkten Automaten (LBA)* akzeptiert wird

2 **kontextfrei**, wenn sie von einem *Kellerautomaten* akzeptiert wird

3 **regulär**, wenn Sie von einem *endlichen Automaten* akzeptiert wird.

[3] engl: recursively enumerable, auch bezeichnet als: Turing-erkennbar (Turing-recognizable), semi-entscheidbar (semi-decidable)

Einschub (WH): Induktive Definition

Gegeben: Grundmenge $A_0 \subseteq B$, Bildungsregel $f\colon B^n \to B$

Stufenweise Konstruktion von Mengen:

$$A_{i+1} = A_i \cup \{f(x_1, \ldots, x_n) \mid x_1, \ldots, x_n \in A_i\}$$

Die dadurch insgesamt definierte Menge ist dann die Vereinigung all dieser Mengen:

$$\mathcal{A} = \bigcup_{i \geq 0} A_i$$

Abgeschlossenheit

Sei B eine Menge und $f\colon B^n \to B$ eine Funktion. Eine Menge $A \subseteq B$ heißt *abgeschlossen unter* f, wenn gilt:

$$x_1, \ldots, x_n \in A \quad \Rightarrow \quad f(x_1, \ldots, x_n) \in A$$

Einschub (WH): Induktive Definition ctd.

Sei B eine Menge, $A_0 \subseteq B$ und $f\colon B^n \to B$. Weiters sein $A_{i+1} = A_i \cup \{f(x_1, \ldots, x_n) \mid x_1, \ldots, x_n \in A_i\}$ sowie $\mathcal{A} = \bigcup_{i \geq 0} A_i$.
Dann gilt:

- $\mathcal{A}$ ist abgegeschlossen unter f.
- Ist $\mathcal{A}'$ abgeschlossen unter f und gilt $A_0 \subseteq \mathcal{A}' \subseteq B$, dann gilt $\mathcal{A} \subseteq \mathcal{A}'$.

D.h.: $\mathcal{A}$ ist die kleinste Menge, die A_0 enthält und abgeschlossen ist unter f.

Schema der induktiven Definition

$\mathcal{A}$ ist die kleinste Menge, für die gilt:

- $A_0 \subseteq \mathcal{A}$
- $x_1, \ldots, x_n \in \mathcal{A} \Rightarrow f(x_1, \ldots, x_n) \in \mathcal{A}$
  ($\mathcal{A}$ ist abgeschlossen unter f)

Reguläre Sprachen

- Gebildet durch Vereinigung, Konkatenation und Stern
- Äquivalent: endliche Automaten, reguläre Grammatiken

Anwendungen in der Informatik:

- Compilerbau: *Tokens* bilden reguläre Sprache, verarbeitet durch Scanner (Lexer).
  Reguläre Ausdrücke dienen als Eingabe für Scannergeneratoren (`lex`, `flex`).
- Texteditoren: erweiterte Suche
- Unix-Shells, `grep`, `awk`, Perl, Xml, ...

Reguläre Sprachen

Reguläre Mengen (Sprachen)

Die Menge $\mathcal{L}_{\text{reg}}(\Sigma)$ der regulären Mengen (Sprachen) über Σ ist die kleinste Menge, sodass

- $\{\}, \{a\} \in \mathcal{L}_{\text{reg}}(\Sigma)$ für alle $a \in \Sigma$
- $A, B \in \mathcal{L}_{\text{reg}}(\Sigma) \Rightarrow A \cup B, A \cdot B, A^* \in \mathcal{L}_{\text{reg}}(\Sigma)$

Reguläre Ausdrücke (Algebraische Notation)

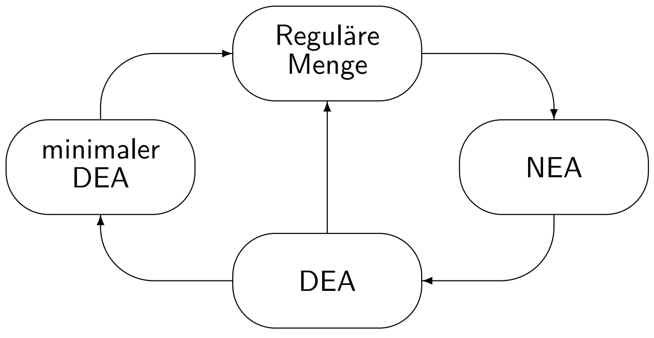
s	statt	$\{s\}$ für $s \in \Sigma$	$L_1 + L_2$	statt	$L_1 \cup L_2$
ε	statt	$\{\varepsilon\}$	$L_1 L_2$	statt	$L_1 \cdot L_2$
$\emptyset$	statt	$\{\}$	L^*	bleibt	L^*

* hat die höchste Priorität, $+$ die niedrigste.

Anmerkung: Genau genommen, ist ein regulärer Ausdruck E keine Sprache. Will man auf die Sprache Bezug nehmen, die von E beschrieben wird, sollte man eigentlich $L(E)$ verwenden.

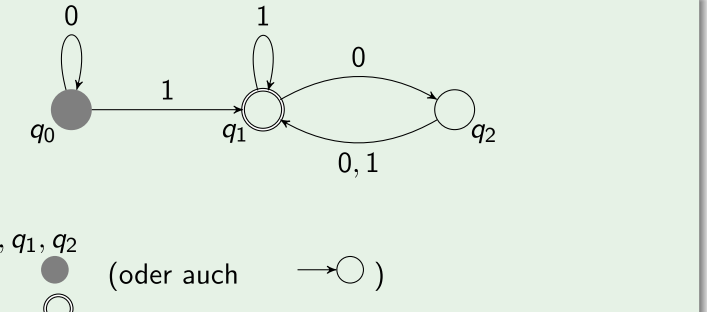
Endliche Automaten

Modell für Systeme mit Ein/Ausgaben aus endlichem Wertebereich und mit endlichem Speicher.



DEA ... deterministischer endlicher Automat
NEA ... nichtdeterministischer endlicher Automat

DEA: Beispiel



Zustände: q_0, q_1, q_2
Startzustand: (oder auch)
Endzustand:
Transitionen (Übergänge): Pfeile

Eingabe: Wort; Ausgehend vom Startzustand liest der Automat $\mathcal{A}$ von links nach rechts Symbol für Symbol. Nach dem Lesen eines Symbols geht $\mathcal{A}$ in den nächsten Zustand über indem er entlang der mit diesem Symbol markierten Kante geht. Nach dem Lesen des letzten Symbols wird der "Output" erzeugt: Befindet sich $\mathcal{A}$ in einem Endzustand, wird das Wort akzeptiert; ansonsten wird das Wort nicht akzeptiert.

Deterministische endliche Automaten (DEA)

DEA

Ein deterministischer endlicher Automat (DEA) ist ein 5-Tupel

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F), \text{wobei}$$

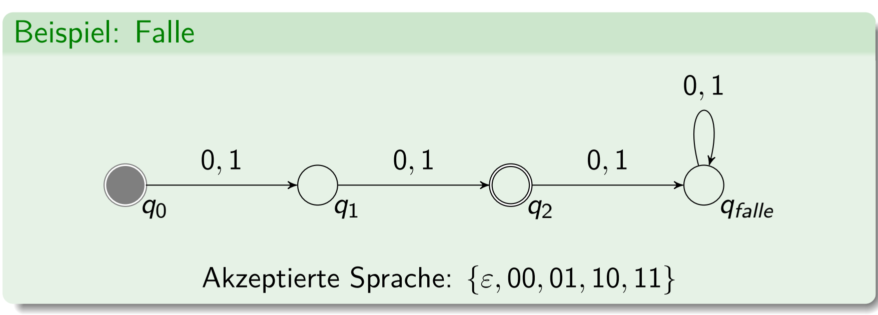
Q ... endliche Menge von Zuständen
Σ ... Eingabealphabet
$\delta\colon Q \times \Sigma \to Q$ ... Übergangsfunktion (total)
$q_0 \in Q$ ... Anfangszustand
$F \subseteq Q$ ... Menge von Endzuständen

DEA: Falle

Falle

Sei $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ ein DEA und $q \in Q - F$ mit $\delta(q, a) = q$ für alle $a \in \Sigma$. Dann heißt q *Falle*.

Beispiel: Falle



Akzeptierte Sprache: $\{\varepsilon, 00, 01, 10, 11\}$

Deterministischer endlicher Automat

Um das Verhalten eines DEA auf einer Zeichenkette formal zu beschreiben, erweitern wir die Übergangsfunktion δ auf beliebige Wörter aus Σ^*:

Erweiterte Übergangsfunktion

$\delta^*\colon Q \times \Sigma^* \to Q$

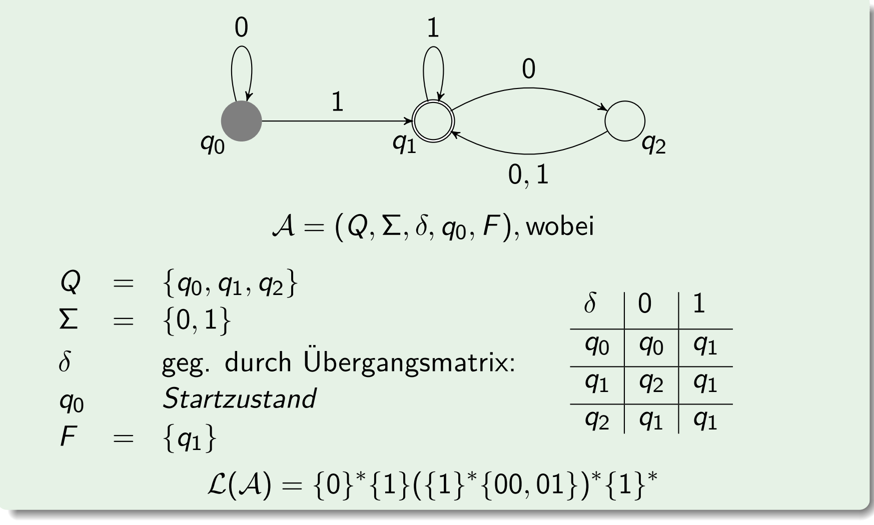
$$\delta^*(q, \varepsilon) = q, \qquad \delta^*(q, aw) = \delta^*(\delta(q, a), w)$$

für alle $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$.

Akzeptierte Sprache

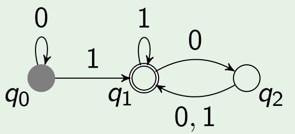
$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$

DEA: Beispiel



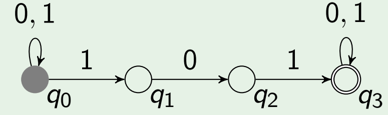
Nichtdeterministische endliche Automaten (NEA)

DEA



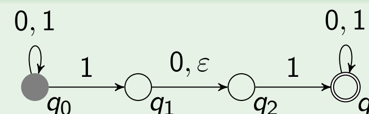
Von einem Zustand aus gibt es mit ein und demselben Eingabesymbol genau einen Folgezustand.

NEA



Von einem Zustand aus kann es mit ein und demselben Eingabesymbol mehrere Folgezustände geben.

ε-NEA



Von einem Zustand aus kann es mit ein und demselben Eingabesymbol mehrere Folgezustände geben, Übergänge sind auch mit ε möglich (ε-Übergänge).

Nichtdeterministische endliche Automaten (NEA)

Übergangsfunktion NEA: $\delta\colon Q \times \Sigma \to \mathcal{P}(Q)$

Übergangsfunktion ε-NEA: $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$

Erweiterte Übergangsfunktion: $\delta^*\colon Q \times \Sigma^* \to \mathcal{P}(Q)$

$$\delta^*(q, w) = \{q' \in Q \mid q \rightsquigarrow wq'\}$$

$q \rightsquigarrow wq'$ ... es gibt einen mit w beschrifteten Pfad von q nach q'

Akzeptierte Sprache:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \{\}\}$$

Automaten $\mathcal{A}$ und $\mathcal{A}'$ sind äquivalent, falls $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

Determinisierung (NEA $\rightarrow$ DEA)

Zu jedem NEA gibt es einen äquivalenten DEA.

Beweis (Teilmengenkonstruktion)

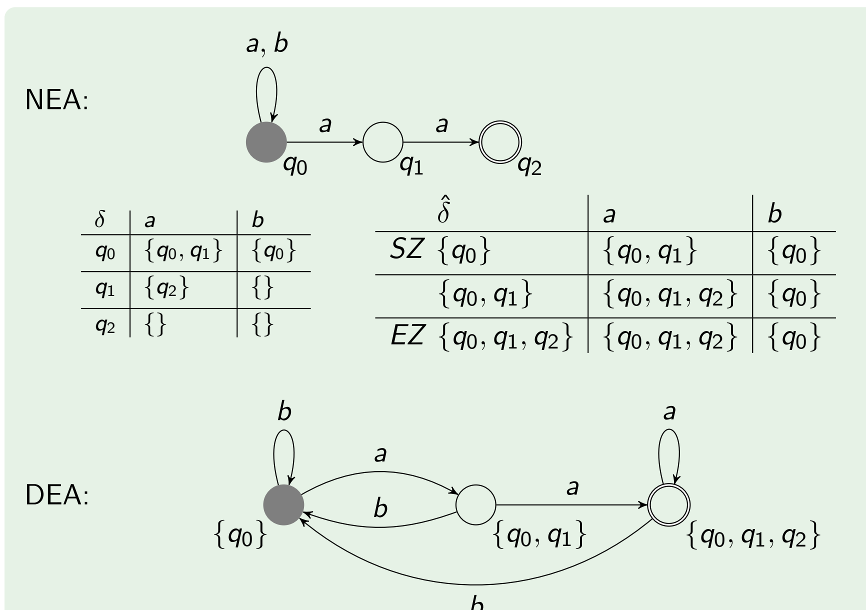
NEA: $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$
DEA: $\hat{\mathcal{A}} = (\hat{Q}, \Sigma, \hat{\delta}, \hat{q}_0, \hat{F})$, wobei

$$
\begin{aligned}
\hat{Q} &= \mathcal{P}(Q) \\
\hat{\delta}(\hat{q}, a) &= \bigcup_{q \in \hat{q}} \delta^*(q, a) \qquad \text{für alle } \hat{q} \in \hat{Q},\ a \in \Sigma \\
\hat{q}_0 &= \{q_0\} \\
\hat{F} &= \begin{cases} \{\hat{q} \in \hat{Q} \mid \hat{q} \cap F \neq \{\}\} \cup \{\hat{q}_0\} & \mathit{falls}\ \varepsilon \in \mathcal{L}(\mathcal{A}) \\ \{\hat{q} \in \hat{Q} \mid \hat{q} \cap F \neq \{\}\} & \mathit{sonst} \end{cases}
\end{aligned}
$$

□

Tipp: Berechne die Übergangsfunktion ausgehend von $\hat{q}_0$ nur für tatsächlich erreichbare Zustände.

Determinisierung: Beispiel

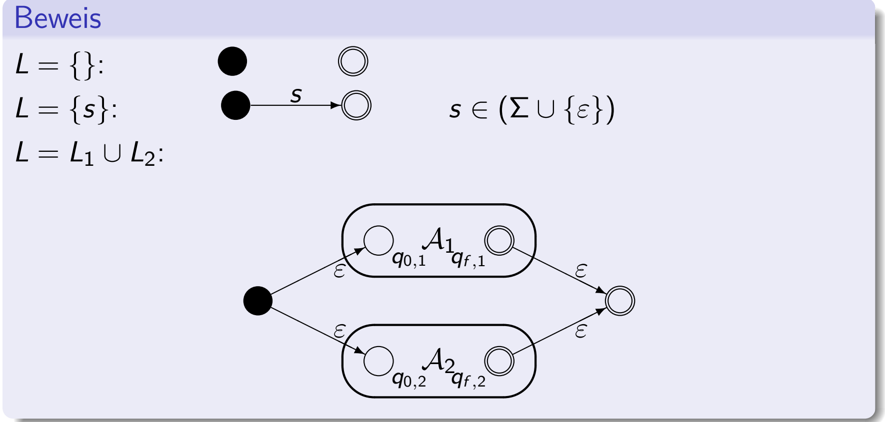


δ	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\{\}$
q_2	$\{\}$	$\{\}$

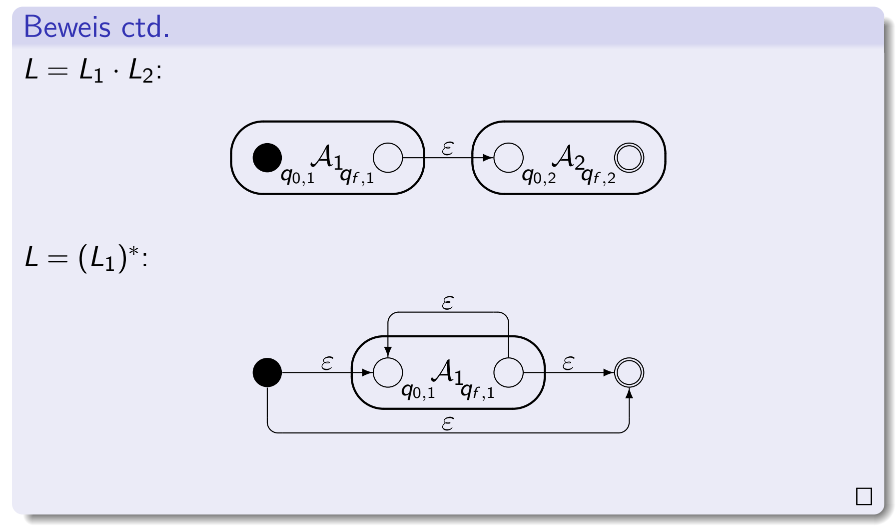
$\hat{\delta}$	a	b
SZ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
EZ $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

Reguläre Menge → ε-NEA

Zu jeder regulären Sprache L gibt es einen endlichen Automaten $\mathcal{A}$, sodass $L = \mathcal{L}(\mathcal{A})$.



Reguläre Menge → ε-NEA



DEA $\to$ reguläre Menge

Jede von einem DEA akzeptierte Sprache ist regulär.

Beweis

$\mathcal{A} = (\{q_1, \ldots, q_n\}, \Sigma, \delta, q_1, F)$
R_{ij}^k ... Menge aller Wörter, mit denen man von q_i nach q_j gelangt, ohne einen Zustand mit einem Index größer als k zu berühren.

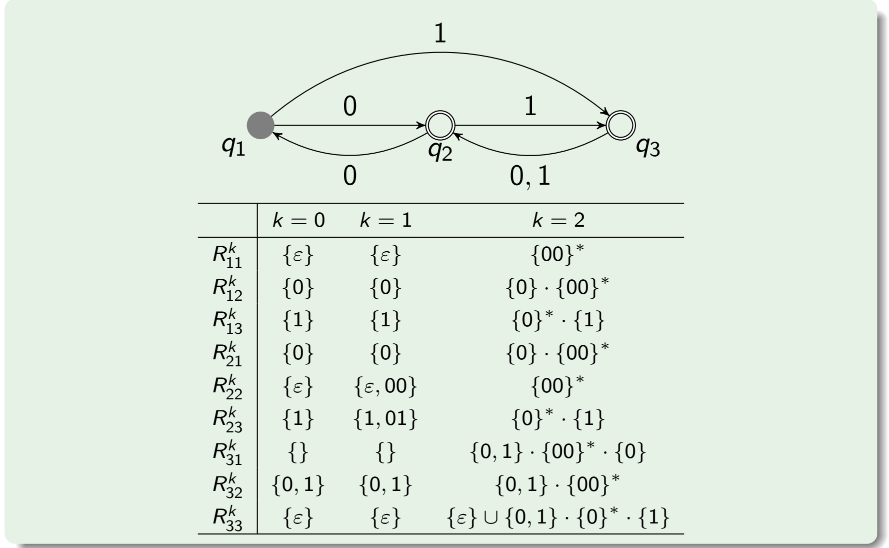
$$R_{ij}^0 = \begin{cases} \{s \in \Sigma \mid \delta(q_i, s) = q_j\} & i \neq j \\ \{s \in \Sigma \mid \delta(q_i, s) = q_j\} \cup \{\varepsilon\} & i = j \end{cases}$$

$$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1} \cdot (R_{kk}^{k-1})^* \cdot R_{kj}^{k-1} \quad k > 0$$

$$\mathcal{L}(\mathcal{A}) = \bigcup_{q_j \in F} R_{1j}^n$$

□

Von DEA zu Regulären Mengen: Beispiel



	$k=0$	$k=1$	$k=2$
R^k_{11}	$\{\varepsilon\}$	$\{\varepsilon\}$	$\{00\}^*$
R^k_{12}	$\{0\}$	$\{0\}$	$\{0\} \cdot \{00\}^*$
R^k_{13}	$\{1\}$	$\{1\}$	$\{0\}^* \cdot \{1\}$
R^k_{21}	$\{0\}$	$\{0\}$	$\{0\} \cdot \{00\}^*$
R^k_{22}	$\{\varepsilon\}$	$\{\varepsilon, 00\}$	$\{00\}^*$
R^k_{23}	$\{1\}$	$\{1, 01\}$	$\{0\}^* \cdot \{1\}$
R^k_{31}	$\{\}$	$\{\}$	$\{0,1\} \cdot \{00\}^* \cdot \{0\}$
R^k_{32}	$\{0,1\}$	$\{0,1\}$	$\{0,1\} \cdot \{00\}^*$
R^k_{33}	$\{\varepsilon\}$	$\{\varepsilon\}$	$\{\varepsilon\} \cup \{0,1\} \cdot \{0\}^* \cdot \{1\}$

Einschub: Graphen

Markierte gerichtete Graphen

Seien U und W endliche Mengen. Ein markierter gerichteter Graph g über U und W ist ein Tripel (K, E, L) wobei

- K die Menge der Knoten
- $E \subseteq K \times W \times K$ die Menge der Kanten
- $L : K \to U$ die Knotenmarkierungsfunktion ist.

Ein Element $(x, w, y) \in E$ stellt eine *gerichtete Kante* vom Knoten x zum Knoten y mit Markierung w dar.

Äquivalenz und Isomorphie

Zwei markierte gerichtete Graphen $g = (K, E, L)$ und $g' = (K', E', L')$ über U und W heißen *isomorph*, wenn wenn es eine bijektive Abbildung $f : K \to K'$ gibt, sodass für alle $k, l \in K$ und $w \in W$ gilt: $(k, w, l) \in E$ genau dann wenn $(f(k), w, f(l)) \in E'$.
Gilt außerdem $L(k) = L'(f(k))$ für alle $k \in K$, so heißen g und g' *äquivalent*.

Minimalautomat

Zu jeder regulären Sprache kann man effektiv einen DEA mit einer minimalen Anzahl von Zuständen konstruieren, der bis auf die Umbenennung der Zustände eindeutig ist.

Minimierungsalgorithmus von Brzozowski[4]

Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein DEA mit $L(A) = L$. Dann ist $A^r = (Q, \Sigma, \{(q, a, p) \mid (p, a, q) \in \delta\}, F, \{q_0\})$ mit $L(A^r) = L^r$.

Minimierungsalgorithmus von Brzozowski

Gegeben ein DEA A. Um den zu A äquivalenten Minimalautomaten C zu erhalten:

1. konstruiere einen NEA A^r durch “Spiegelung” von A
2. konstruiere einen DEA B durch Determinisierung von A^r
3. konstruiere einen NEA B^r durch “Spiegelung” von B
4. konstruiere einen DEA C durch Determinisierung von B^r

[4] Janusz (John) Antoni Brzozowski (*1935)

Abschlusseigenschaften regulärer Sprachen

$\mathcal{L}_{\mathrm{reg}}(\Sigma)$ ist **abgeschlossen** gegenüber:

- Vereinigung, Verkettung, Stern-Operator: per Definition.
- Plus-Operator: $A^+ = A^* \cdot A$.
- Komplement bzgl. Σ^*: konstruiere DEA und vertausche End- und Nichtendzustände. (Falle nicht vergessen!)
- Durchschnitt: $A \cap B = \overline{\overline{A} \cup \overline{B}}$.
- Differenz: $A - B = A \cap \overline{B}$.
- Spiegelung: Konstruiere EA, vertausche Start- mit Endzustand, kehre alle Zustandsübergänge um.
- Homomorphismen: repräsentiere L durch regulären Ausdruck und ersetze alle Vorkommnisse von a durch Ausdruck für $h(a)$.

Homomorphismus

Seien Σ und Γ zwei Alphabete. **Homomorphismus**: $h : \Sigma \longrightarrow \Gamma^*$
Induktive Erweiterung auf Wörter über Σ:

1. $h(\varepsilon) = \varepsilon$
2. $h(wa) = h(w)h(a)$ für alle $w \in \Sigma^*$ und $a \in \Sigma$.

Anwendung von h auf jedes Wort der Sprache L:

$$h(L) = \{h(w) \mid w \in L\}$$

Beispiel

Sei $h : \{0,1\}^* \longrightarrow \{a,b\}^*$ ein Homomorphismus mit $h(0) = ab$ und $h(1) = \varepsilon$

Dann ist $h(0011) = abab$ und $h(L(10^*1)) = L((ab)^*)$

Entscheidungsprobleme

Folgende Probleme sind für reguläre Sprachen L, L' **entscheidbar**:

- Gehört ein Wort w der Sprache L an? (**Wortproblem**)
  Konstruiere einen DEA für L und prüfe, ob $\delta^*(q_0, w) \in F$.
- Ist L leer?
  Konstruiere einen DEA für L und prüfe, ob von q_0 aus ein Endzustand erreichbar ist.
- Ist L endlich oder unendlich?
  L ist unendlich gdw. der minimale DEA für L (abgesehen von der Falle!) einen Zyklus enthält.
- Gilt $L = L'$?
  Überprüfe, ob $L - L'$ und $L' - L$ leer sind.

Grenzen der Regularität

Sei $L = \{0^n 1^n \mid n \geq 0\}$. Gibt es einen DEA für L?

Es sieht so aus als müsste sich die Maschine die Anzahl der Nullen merken. Nachdem ebendiese Anzahl aber nicht limitiert ist, müsste sich der DEA eine unbeschränkte Anzahl von Möglichkeiten merken können. Mit einer endlichen Anzahl von Zuständen ist das aber nicht möglich!

Aber: Nur weil es so aussieht, als ob eine Sprache unbegrenzten Speicher brauchen würde, ist das nicht notwendigerweise so!

Regulär oder nicht?

$A = \{w \mid w \text{ enthält gleich viele Symbole 0 wie Symbole 1}\}$
$B = \{w \mid \text{die Teilwörter}^5 \text{ 01 und 10 kommen gleich oft in } w \text{ vor}\}$

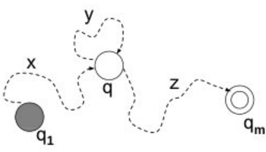
[5] Für ein Wort $w \in \Sigma^*$, wobei $w = xuy$ für Wörter $x, u, y \in \Sigma^*$ heißt u **Teilwort** von w.

Schubfachprinzip[6]

Schubfachprinzip (pigeonhole principle)

Bei Verteilung von $n+1$ Gegenständen auf n Schubfächer müssen in mindestens einem Schubfach zwei Gegenstände landen.

DEA $\mathcal{A}$ habe m Zustände. Gilt für ein von $\mathcal{A}$ akzeptiertes Wort w, dass $|w| \geq m$, so muss mindestens ein Zustand mehr als einmal durchlaufen werden. (Schubfachprinzip!)
Zerlege w in xyz: y gehe von q nach q; diese Schleife kann ausgelassen bzw. beliebig oft wiederholt werden, d.h. xy^iz wird für alle i ebenfalls von $\mathcal{A}$ akzeptiert!



[6] Johann Peter Gustav Lejeune Dirichlet, 1805-1859

Pumping Lemma für reguläre Sprachen

(Unendliche) Reguläre Sprachen haben eine spezielle Eigenschaft: Jedes Wort ab einer bestimmten Länge kann "aufgepumpt" werden, d.h. jedes solche Wort enthält ein Teilstück, das beliebig oft wiederholt werden kann, wobei die resultierenden Wörter noch immer in derselben Sprache liegen.

Pumping Lemma für reguläre Sprachen

Sei L eine unendliche reguläre Sprache. Dann **gibt es** eine (nur von L abhängige) Schranke $m > 0$ so, dass für **jedes** Wort $w \in L$ mit $|w| \geq m$ Wörter x, y, z so **existieren**, dass

$$w = xyz \qquad \text{mit } |xy| \leq m \text{ und } |y| > 0$$

sowie

$$w_i = xy^i z \in L \qquad \textbf{für alle } i \geq 0.$$

Reguläres Pumping Lemma - Beispiel

Zu zeigen: $L = \{a^n b^n \mid n \geq 0\}$ ist nicht regulär.

Beweis durch Widerspruch:

Angenommen L sei regulär. Für **beliebiges** m (Konstante aus dem Pumping Lemma) wähle **ein** $w \in L$ mit $|w| \geq m$, z.B.

$$w = a^m b^m$$

Betrachte **beliebige** Zerlegungen von w in xyz, wobei $|xy| \leq m$ und $|y| > 0$: xy kann nur aus Symbolen a bestehen.

Wähle **ein** i so, dass $xy^i z$ nicht von der Gestalt $a^n b^n$ ist:

z.B. $i = 2$, dann müsste – wäre L regulär – auch $xy^2 z = a^{m+|y|} b^m$ aus L sein, was aber nicht der Fall ist! Widerspruch! L kann nicht regulär sein.

Grammatiken: Formale Definition

Grammatik

Grammatik $G = (N, T, P, S)$, wobei

N ... endliche Menge von Nonterminalen (Variablen)
T ... endliche Menge von Terminalsymbolen ($N \cap T = \{\}$)
$P \subseteq (N \cup T)^+ \times (N \cup T)^*$ ... Produktionen
$S \in N$ ... Startsymbol

Wir schreiben
$\alpha \to \beta$ statt $(\alpha, \beta) \in P$
$\alpha \to \beta_1 \mid \cdots \mid \beta_n$ statt $\alpha \to \beta_1, \ldots, \alpha \to \beta_n$

Ableitungen

Direkte Ableitung:

$x\alpha y \Rightarrow x\beta y$ falls $\alpha \to \beta \in P$ und $x, y \in (N \cup T)^*$

Ableitung von w aus v **in** n **Schritten**:

Es gibt Wörter $w_0, w_1, \ldots, w_n$, sodass $v = w_0$, $w = w_n$ und $w_{i-1} \Rightarrow w_i$ für $1 \leq i \leq n$.
Schreibweise: $v \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_{n-1} \Rightarrow w$ bzw. $v \Rightarrow^n w$

Ableitung in mehreren Schritten:

$v \Rightarrow^* w$ ... reflexiver und transitiver Abschluss von $\Rightarrow$

Satzformen und erzeugte Sprache

Gilt $S \Rightarrow w$ für ein Wort $w \in (N \cup T)^*$, so nennt man w **Satzform**.

Menge aller in n Schritten ableitbaren Satzformen:
$SF(G, n) = \{w \in (N \cup T)^* | S \Rightarrow^n w\}$

Erzeugte Sprache

Von G erzeugte *Sprache* $\mathcal{L}(G)$:

$$\mathcal{L}(G) \quad = \quad \{w \in T^* \mid S \Rightarrow^* w\}$$

Grammatiken G_1 und G_2 heißen äquivalent, wenn $\mathcal{L}(G_1) = \mathcal{L}(G_2)$ gilt.

Grammatiken: Beispiele

$\{a^n \mid n \geq 0\}$

$$G_1 = (\{S\}, \{a\}, \{S \to \varepsilon, S \to aS\}, S)$$

$$\mathcal{L}(G_1) = \{\varepsilon\} \cup \{a^n \mid n \geq 1\} = \{a\}^*$$

Alle in G_1 möglichen Ableitungen sind von der Gestalt

$$S \Rightarrow \varepsilon \quad \text{bzw.} \quad S \Rightarrow aS \Rightarrow^n a^{n+1}S \Rightarrow a^{n+1}$$

für ein $n \geq 0$.

Grammatiken: Beispiele

$\{a^n b^n | n \geq 0\}$

$$G_2 = (\{S\}, \{a, b\}, \{S \rightarrow aSb, S \rightarrow \varepsilon\}, S)$$

$$\mathcal{L}(G_2) = \{a^n b^n | n \geq 0\}$$

Alle in G_2 möglichen Ableitungen sind von der Gestalt

$$S \Rightarrow^n a^n S b^n \Rightarrow a^n b^n \text{ für alle } n \geq 0$$

Formaler Beweis mittels *Induktion*:
Menge aller Satzformen nach genau $n \geq 1$ Schritten: $SF(G_2, n) = \{a^n S b^n, a^{n-1} b^{n-1}\}$

Induktionsbasis:	$SF(G_2, 1) = \{a^1 S b^1, \varepsilon\}$
Induktionshypothese:	$SF(G_2, n) = \{a^n S b^n, a^{n-1} b^{n-1}\}$
Induktionsbehauptung:	$SF(G_2, n+1) = \{a^{n+1} S b^{n+1}, a^n b^n\}$

Beweis: Das Wort $a^{n-1} b^{n-1}$ ist terminal und daher nicht mehr weiter ableitbar. Aus $a^n S b^n$ ist ableitbar:

mittels $S \rightarrow aSb$: $a^{n+1} S b^{n+1}$ mittels $S \rightarrow \varepsilon$: $a^n b^n$

Grammatiken: Beispiele

$\{a^n b^n c^n | n \geq 1\}$

$$G_3 = (\{S, A, C\}, \{a, b, c\}, P_3, S)\text{wobei}$$

$$P_3 = \{S \to abc, S \to aAbc, A \to aAbC, A \to abC, Cb \to bC, Cc \to cc\}$$

$$\mathcal{L}(G_3) = \{a^n b^n c^n | n \geq 1\}$$

Alle in G_3 möglichen Ableitungen sind von der Gestalt $S \Rightarrow abc$ bzw. für $n \geq 2$:

$$S \Rightarrow aAbc \Rightarrow^{n-2} a^{n-1}A(bC)^{n-2}bc \Rightarrow a^n(bC)^{n-1}bc \Rightarrow^* a^n b^n c^n$$

Grammatik-Typen: Typ-i-Grammatiken

Typ-i-Grammatiken

Sei $G = \langle N, T, P, S \rangle$ eine Grammatik. Dann heißt G auch **unbeschränkte Grammatik** (**Typ-0**).
Gilt für alle Produktionen $\alpha \to \beta \in P$

- $|\alpha| \leq |\beta|$ so heißt G **monoton**;
- $\alpha = uAv$ und $\beta = uwv$ für ein $A \in N$, $w \in (N \cup T)^+$ und $u, v \in (N \cup T)^*$ so heißt G **kontextsensitiv** (**Typ-1**);
- $A \to \beta$ für ein $A \in N$, so heißt G **kontextfrei** (**Typ-2**);
- $A \to aB$ oder $A \to \varepsilon$ für $A, B \in N$ und $a \in T$, so heißt G **regulär** (**Typ-3**).

Für monotone und kontextsensitive Grammatiken gilt: Kommt S nicht auf der rechten Seite einer Produktion vor, so ist auch die Produktion $S \to \varepsilon$ erlaubt.

Erzeugte Sprachen

Erzeugte Sprachen

Eine formale Sprache heißt **rekursiv aufzählbar**, **monoton**, **kontextsensitv**, **kontextfrei** bzw. **regulär**, wenn sie von einer Typ-0-, monotonen, Typ-1-, Typ-2-, bzw. Typ-3-Grammatik erzeugt wird.

Aufgrund der Definition können wir nun die einzelnen Sprachen aus den vorigen Beispielen klassifizieren:

Es ergibt sich, dass
$\mathcal{L}(G_1) = \{a^n | n \geq 0\}$ regulär
$\mathcal{L}(G_2) = \{a^n b^n | n \geq 0\}$ kontextfrei und
$\mathcal{L}(G_3) = \{a^n b^n c^n | n \geq 1\}$ monoton ist.

Reguläre Grammatiken

Reguläre Grammatiken [1]

Eine Grammatik heißt *regulär*, wenn alle Produktionen von der Form

$$A \to aB \qquad \text{oder} \qquad A \to \varepsilon$$

sind ($A, B \in N$, $a \in T$).

Alternativ:

Reguläre Grammatiken [2]

Eine Grammatik heißt *regulär*, wenn alle Produktionen von der Form

$$A \to aB \qquad \text{oder} \qquad A \to a$$

sind ($A, B \in N$, $a \in T$). Um auch das Leerwort erzeugen zu können, ist $S \to \varepsilon$ erlaubt, sofern S nicht auf der rechten Seite einer Produktion vorkommt.

Reguläre Grammatiken: Beispiele

$L = \{a\}^*$

$$G_1 = (\{S\}, \{a\}, \{S \to aS \mid \varepsilon\}, S) \quad \text{(Def. [1])}$$
$$G_2 = (\{S, T\}, \{a\}, \{S \to aT \mid a \mid \varepsilon, T \to aT \mid a\}, S) \quad \text{(Def. [2])}$$

$$\mathcal{L}(G_1) = \mathcal{L}(G_2) = L$$

$L = \{a\}^+$

$$G_1 = (\{S, T\}, \{a\}, \{S \to aT, T \to aT \mid \varepsilon\}, S) \quad \text{(Def. [1])}$$
$$G_2 = (\{S\}, \{a\}, \{S \to aS \mid a\}, S) \quad \text{(Def. [2])}$$

$$\mathcal{L}(G_1) = \mathcal{L}(G_2) = L$$

Reguläre Grammatiken und Sprachen

Eine Sprache ist genau dann regulär, wenn sie von einer regulären Grammatik erzeugt wird. Umgekehrt lässt sich zu jeder regulären Sprache eine reguläre Grammatik finden, die sie generiert.

Beweis:
Simuliere DEA mittels regulärer Grammatik und umgekehrt.

DEA und reguläre Grammatik

DEA: $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$

definiere reguläre Grammatik

$$G = (Q, \Sigma, P, q_0)$$

wobei P wie folgt definiert wird:

$$\begin{array}{ll} q \to ap \in P & \text{wenn } \delta(q, a) = p \\ q \to \varepsilon \in P & \text{wenn } q \in F \end{array}$$

Es folgt dann: $\delta^*(q_0, w) \in F$ g.d.w. $q_0 \Rightarrow^* w$

□

DEA und reguläre Grammatik: Beispiel

$\mathcal{A} = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$ mit

$$\delta(q_0, 0) = q_0,\ \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_1,\ \delta(q_1, 1) = q_0$$

Definiere $G = (\{q_0, q_1\}, \{0, 1\}, P, q_0)$ mit

$$\begin{aligned} P \quad = \quad & \{q_0 \to 0\ q_0 \mid 1\ q_1, \\ & \ \ q_1 \to 0\ q_1 \mid 1\ q_0 \mid \varepsilon\} \end{aligned}$$

Ableitung von 0 1 0:

$$q_0 \Rightarrow 0\ q_0 \Rightarrow 0\ 1\ q_1 \Rightarrow 0\ 1\ 0\ q_1 \Rightarrow 0\ 1\ 0$$

Kontextfreie Grammatiken

Kontextfreie Grammatik

Eine Grammatik heißt *kontextfrei*, wenn alle Produktionen von der Form $A \to \beta$ sind, wobei $A \in N$ und $\beta \in (N \cup T)^*$.
Eine Sprache L heißt kontextfrei, wenn es eine kontextfreie Grammatik G gibt, sodass $L = \mathcal{L}(G)$.

Linksableitung:

$xAy \Rightarrow_L x\beta y$ falls $A \to \beta \in P$ und $x \in T^*$

Rechtsableitung:

$xAy \Rightarrow_R x\beta y$ falls $A \to \beta \in P$ und $y \in T^*$

Parallelableitung:

$x_0 A_1 x_1 \cdots A_n x_n \Rightarrow_P x_0 \beta_1 x_1 \cdots \beta_n x_n$ falls
$A_1 \to \beta_1, \ldots, A_n \to \beta_n \in P$ und $x_0, \ldots, x_n \in T^*$

Ableitungsvarianten

Alle Ableitungsvarianten ergeben dieselbe Sprache:

Ist G eine kontextfreie Grammatik, dann gilt:

$$
\begin{aligned}
\mathcal{L}(G) &= \{w \in T^* \mid S \Rightarrow^* w\} \\
&= \{w \in T^* \mid S \Rightarrow_L^* w\} \\
&= \{w \in T^* \mid S \Rightarrow_R^* w\} \\
&= \{w \in T^* \mid S \Rightarrow_P^* w\}
\end{aligned}
$$

Kontextfreie Grammatiken: Beispiele

Wohlgeformte Klammerausdrücke

WKA ist die kleinste Menge, sodass

- $\varepsilon \in WKA$
- $w \in WKA \Rightarrow (w) \in WKA$
- $w_1, w_2 \in WKA \Rightarrow w_1 w_2 \in WKA$

$$G = (\{A\}, \{(,)\}, \{A \to \varepsilon \mid (A) \mid A\,A\}, A)$$

Kontextfreie Grammatiken: Beispiele

Palindrome über $\{a, b\}$

$$G = (\{S\}, \{a, b\}, \{S \to \varepsilon \mid a \mid b \mid a\,S\,a \mid b\,S\,b\}, S)$$

$$\mathcal{L}(G) = \{\varepsilon, a, b, aa, bb, aaa, aba, bab, bbb, \ldots\}$$

$L = \{w \in \{a, b\}^* \mid |w|_a = 2 \cdot |w|_b\}$

$G = (\{S\}, \{a, b\}, P, S)$, wobei

$$P = \{S \to a\,S\,a\,S\,b\,S \mid a\,S\,b\,S\,a\,S \mid b\,S\,a\,S\,a\,S \mid \varepsilon\}$$

Kontextfreie Grammatiken: Beispiele

$L = \{a^m + b^n = c^{m+n} \mid m, n \geq 0\}$

Kontextfreie Grammatik $G = (\{S, T\}, \{a, b, c, +, =\}, P, S)$
wobei P:

$$P = \{ \begin{array}{lll} S & \rightarrow & a\,S\,c \mid +\,T \\ T & \rightarrow & b\,T\,c \mid = \end{array} \}$$

$aa + bb = cccc \in \mathcal{L}(G)$:

$$\begin{array}{llllll} S & \Rightarrow & a\,S\,c & \Rightarrow & aa\,S\,cc & \Rightarrow & aa+\,T\,cc \\ & \Rightarrow & aa + b\,T\,ccc & \Rightarrow & aa + bb\,T\,cccc & \Rightarrow & aa + bb = cccc \end{array}$$

Einschub: Bäume

Baum

Ein (geordneter, markierter gerichteter) *Baum* über U und W ist ein Graph $g = (K, E, L)$ über U und W, der folgende Bedingungen erfüllt:

1. $W = \{1, ..., n\}$ für ein $n \geq 1$.
2. Es gibt genau einen ausgezeichneten Knoten p_0 (Wurzel), der keinen Vorgänger hat. Außerdem gibt es von der Wurzel aus zu jedem anderen Knoten q von g einen Pfad $(p_0, e_1, p_1, \ldots, p_{k-1}, e_k, q)$ der Länge $k \geq 1$, $(p_i, e_{i+1}, p_{i+1}) \in E$, $0 \leq i < k$, $q = p_k$.
3. Jeder von der Wurzel verschiedene Knoten hat genau einen Vorgänger.
4. Jeder Knoten ohne Nachfolger heißt Blatt.
5. Ist p kein Blatt, so sind die Nachfolger von p geordnet (die Kanten tragen die Bezeichnungen 1 bis k für ein k).

Ableitungsbäume für kontextfreie Grammatiken

Ableitungsbaum

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Ein Baum $g = (K, E, L)$ über $U = N \cup T \cup \{\varepsilon\}$ und $W = \{1, ..., n\}$ heißt *Ableitungsbaum* für G, wenn Folgendes gilt:

1. Ist p_0 die Wurzel von g, so gilt $L(p_0) = S$.
2. Ist p kein Blatt, so muss $L(p) \in N$ gelten.
3. Ist p ein Blatt mit $L(p) = \varepsilon$, so ist p der einzige Nachfolger seines Vorgängers.
4. Ist $\{(p, i, q_i) \mid i \in 1, ..., n\}$ die geordnete Menge der von p mit $L(p) = A$ wegführenden Kanten, so ist $A \to L(q_1) \ldots L(q_k)$ eine Produktion in P.

A-Baum für G: für Wurzel gilt $L(p_0) = A$
Ein Ableitungsbaum ist ein S-Baum für G.

Front eines Ableitungsbaumes

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik und $g = (K, E, L)$ ein A-Baum für G.

Wir definieren nun eine Ordnungsrelation auf den Pfaden von g:
Seien $P(j) = (p_{j,0}, e_{j,1}, p_{j,1}, \ldots, e_{j,k_j}, p_{j,k_j})$ für $j \in \{1, 2\}$ zwei voneinander verschiedene Pfade in g, die in der Wurzel beginnen (i.e., $p_{1,0} = p_{2,0} = p_0$) und zu einem Blatt von g führen, dann definieren wir $P_1 < P_2$ genau dann, wenn es ein $m \geq 1$ so gibt, dass $e_{1,i} = e_{2,i}$ für alle $1 \leq i < m$ und $e_{1,m} < e_{2,m}$.

Betrachten wir nun alle derartigen Pfade in g, so sind diese wohlgeordnet und sind $p_1, \ldots, p_k$ die Blätter dieser Pfade, so ist die **Front** von g durch $L(p_1) \ldots L(p_k)$ definiert.

Ableitungen und Ableitungsbäume

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik und $g = (K, E, L)$ ein A-Baum für G sowie $w \in (N \cup T)^*$.

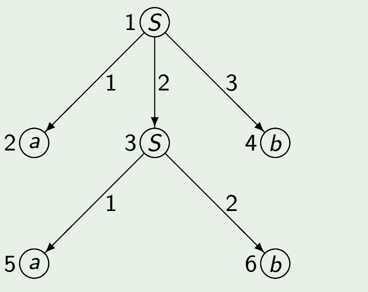
Dann gilt $A \Rightarrow^* w$ genau dann, wenn es einen A-Baum für G mit Front w gibt. Somit gilt für alle $w \in T^*$ auch $S \Rightarrow^* w$ genau dann, wenn es einen Ableitungsbaum für G mit Front w gibt.

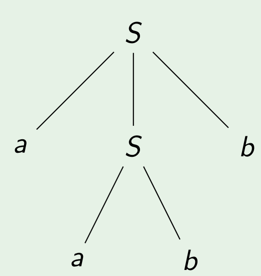
Jeder Linksableitung in G kann man eindeutig einen Ableitungsbaum zuordnen. Gibt es zwei verschiedene Linksableitungen in G für ein Wort w, so sind die entsprechenden Ableitungsbäume *nicht* äquivalent (im Sinne der Äquivalenz von Graphen).

Ableitungsbäume: Beispiel

$G = (\{S\}, \{a, b\}, \{S \to aSb, S \to ab\}, S)$

Ableitungsbaum für G:





Front des Ableitungsbaums:
ergibt sich als Folge der Labels der Knoten 2, 5, 6, 4 zu *aabb*

TIME FLIES LIKE AN ARROW
FRUIT FLIES LIKE A BANANA

Eindeutigkeit, (inhärente) Mehrdeutigkeit

Sei $G = \langle N, T, P, S \rangle$ eine kontextfreie Grammatik. G heißt **eindeutig**, wenn es zu jedem in G ableitbaren Terminalwort genau eine Linksableitung in G gibt. Ansonsten heißt G **mehrdeutig**.

Eine kontextfreie *Sprache* L heißt **inhärent mehrdeutig**, wenn jede Grammatik, die L erzeugt, mehrdeutig ist.

Mehrdeutigkeit: Beispiel

If-then-else Anweisung

$G_1 = (\{Anw\}, \{if, then, else, expr, others\}, P_1, Anw)$

wobei P_1 folgende Produktionen enthält:

$$
\begin{array}{rcl}
Anw & \rightarrow & if\ expr\ then\ Anw \\
 & | & if\ expr\ then\ Anw\ else\ Anw \\
 & | & others
\end{array}
$$

G_1 ist mehrdeutig, da das Wort

$$if\ expr\ then\ if\ expr\ then\ others\ else\ others$$

zwei Linksableitungen besitzt.

Mehrdeutigkeit: Beispiel ctd.

$\mathcal{L}(G_1)$ ist aber nicht mehrdeutig, da es z.B. folgende eindeutige Grammatik mit $\mathcal{L}(G_1) = \mathcal{L}(G_2)$ gibt:

$$G_2 = (\{Anw, AnwT, AnwTE\},\\ \{if, then, else, expr, others\}, P_2, Anw)$$

wobei P_2 folgende Produktionen enthält:

$$\begin{array}{lll} Anw & \rightarrow & AnwT \\ & | & AnwTE \\ AnwT & \rightarrow & if\ \ expr\ \ then\ Anw \\ & | & if\ \ expr\ \ then\ AnwTE\ \ else\ \ AnwT \\ AnwTE & \rightarrow & if\ \ expr\ \ then\ AnwTE\ \ else\ \ AnwTE \\ & | & others \end{array}$$

Inhärent mehrdeutige Sprachen: Beispiel

Die kontextfreie Sprache $L = L_1 \cup L_2$ mit

$L_1 = \{a^m b^m c^n \mid m, n \geq 1\}$ und $L_2 = \{a^m b^n c^n \mid m, n \geq 1\}$

ist inhärent mehrdeutig.
L ist kontextfrei:

$$
\begin{array}{lcllcl}
S & \to & S_1 \mid S_2 & & & \\
S_1 & \to & S_1\, c \mid A & S_2 & \to & a\, S_2 \mid B \\
A & \to & a\, A\, b \mid \varepsilon & B & \to & b\, B\, c \mid \varepsilon
\end{array}
$$

Grammatik ist mehrdeutig, da $a^n b^n c^n$ zwei verschiedene Ableitungen besitzt.
Man kann zeigen, dass *alle* Grammatiken für L mehrdeutig sind, d.h., L ist inhärent mehrdeutig.

Bemerkung: $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$

Transformation von Produktionen

Nutzlose Produktionen: Entfernen aller Produktionen, die vom Startsymbol aus nicht erreichbar sind oder aus denen kein $w \in T^*$ ableitbar ist.

ε-**Produktionen:** Entfernen aller Produktionen $A \to \varepsilon$

Einheitsproduktionen: Entfernen aller Produktionen $A \to B$

Zu jeder kontextfreien Sprache ohne Leerwort gibt es eine kontextfreie Grammatik ohne nutzlose, ohne ε- und ohne Einheitsproduktionen.

Vorbereitende Schritte zur Chomsky NF

Ausgehend von $G = \langle N, T, P, S \rangle$ konstruieren wir k.f. Grammatiken $G_i = \langle N_i, T_i, P_i, S \rangle$, $1 \leq i \leq 4$, sodass:

1. aus jeder Variablen $A \in N$ ein Terminalwort $w \in T^*$ ableitbar ist,
2. für jedes Symbol $a \in (N \cup T)$ eine Satzform v existiert, die a enthält,
3. P keine ε-Produktionen enthält, also keine Produktionen der Gestalt $A \to \varepsilon$, und
4. P keine Produktionen der Gestalt $A \to B$ (sogenannte *Einheitsproduktionen*), für Variablen $A, B \in N$, enthält.

1. Entfernen nutzloser Produktionen (Teil 1)

Menge aller Variablen, aus denen direkt ein *Terminalwort ableitbar* ist:

$$N_1^{(1)} = \{A \in N \mid A \to w \in P, w \in T^*\}$$

Für $i > 1$ definieren wir iterativ

$$N_1^{(i)} = \{A \in N \mid A \to w \in P, w \in (N_1^{(i-1)} \cup T)^*\} \cup N_1^{(i-1)}$$

Sobald $N_1^{(m+1)} = N_1^{(m)}$ (für ein $m > 1$), erhalten wir
$G_1 = \langle N_1, T, P_1, S \rangle$, wobei
$N_1 = N_1^{(m)}$
P_1 enthält alle Produktionen aus P, welche nur Symbole aus $N_1 \cup T$ enthalten.

Bemerkung: Wendet man diese Konstruktion auf eine beliebige kontextfreie Grammatik G an, so erhält man sofort einen *Entscheidungsalgorithmus* für das Problem ob die von G erzeugt Sprache leer ist oder nicht:

Denn $L(G)$ ist genau dann nicht leer, wenn aus der Startvariablen ein Terminalwort ableitbar ist (oder umgekehrt: $L(G) = \{\}$ wenn $S \notin N_1^{(m)}$).

Vorbereitende Schritte zur Chomsky NF: Beispiel

Schritt 1

$G = \langle \{S, A, B, C, D\}, \{a\}, P, S\rangle$ mit

$P = \{S \to aA \mid B \mid D, A \to aB, B \to A, B \to \varepsilon, C \to a\}$

Wir untersuchen, ob aus jeder Variablen in G ein Terminalwort ableitbar ist:

- $N_1^{(1)} = \{B, C\}$
- $N_1^{(2)} = \{S, A\} \cup \{B, C\} = N_1^{(3)}$.

Aus D ist kein Terminalwort ableitbar, wir erhalten also

$$\begin{aligned} G_1 &= \langle N_1, \{a\}, P_1, S\rangle \text{ mit} \\ N_1 &= \{S, A, B, C\} \\ P_1 &= \{S \to aA \mid B, A \to aB, B \to A, B \to \varepsilon, C \to a\} \ . \end{aligned}$$

Nun ist in G_1 aus jeder Variablen ein Terminalwort ableitbar und es gilt $\mathcal{L}(G_1) = \mathcal{L}(G)$.

2. Entfernen nutzloser Produktionen (Teil 2)

Ausgehend von G_1 bestimmen wir nun die Menge aller Symbole V, die vom Startsymbol S aus *erreichbar* sind:
$V_2^{(1)} = \{S\}$
sowie für $i > 1$

$$\begin{array}{rcl} V_2^{(i)} & = & \{\alpha \in N_1 \cup T \mid A \to u\alpha v \in P_1, A \in (V_2^{(i-1)} \cap N_1), \\ & & \quad u, v \in (N_1 \cup T)^*\} \\ & \cup & V_2^{(i-1)} \end{array}$$

Sobald $V_2^{(m+1)} = V_2^{(m)}$, erhalten wir $G_2 = \langle N_2, T_2, P_2, S\rangle$ (mit $\mathcal{L}(G_2) = \mathcal{L}(G)$), wobei
$N_2 = V_2^{(m)} \cap N_1$, sowie
$T_2 = V_2^{(m)} \cap T_1$ und
P_2 enthält alle Produktionen, die nur Symbole aus $N_2 \cup T_2$ enthalten.

Vorbereitende Schritte zur Chomsky NF: Beispiel ctd.

Schritt 2

Ausgehend von G_1 überprüfen wir, ob alle Symbole vom Startsymbol S aus erreichbar sind:

- $V_2^{(1)} = \{S\}$
- $V_2^{(2)} = \{A, B, a\} \cup \{S\} = V_2^{(3)}$

C nicht vom Startsymbol erreichbar. Daher erhalten wir

$$\begin{aligned} G_2 &= \langle N_2, \{a\}, P_2, S\rangle \text{ mit} \\ N_2 &= \{S, A, B\} \\ P_2 &= \{S \to aA \mid B, A \to aB, B \to A, B \to \varepsilon\} \ . \end{aligned}$$

Mit G_2 haben wir nun also eine zu G äquivalente Grammatik ohne nutzlose Symbole und es gilt $\mathcal{L}(G_2) = \mathcal{L}(G)$.

3. Elimination der ε-Produktionen

Ausgehend von der zu G äquivalenten Grammatik G_2 ohne nutzlose Symbole, bestimmen wir nun die Menge aller Variablen, aus denen das Leerwort ableitbar ist (n_2 bezeichnet die Anzahl der Symbole in N_2):

$$N_3^{(1)} = \{A \in N_2 \mid A \to \varepsilon \in P_2\}.$$

Für i mit $1 < i \leq n_2$ definieren wir iterativ

$$N_3^{(i)} = \{A \in N_2 \mid A \to w \in P_2, w \in (N_3^{(i-1)})^*\} \cup N_3^{(i-1)}.$$

Klarerweise gilt, dass aus einer Variablen A genau dann das Leerwort ableitbar ist, wenn $A \in N_3^{(n_2)}$. Daher erhält man sofort für die Startvariable S, dass $\varepsilon \in \mathcal{L}(G_2)$ $(= \mathcal{L}(G))$ genau dann, wenn $S \in N_3^{(n_2)}$.

3. Elimination der ε-Produktionen ctd.

Wir entfernen nun alle ε-Produktionen aus P_2 und nehmen anstelle einer Produktion $A \to X_1...X_k$ mit $X_i \in N_2 \cup T$, $1 \leq i \leq k, k \geq 1$, aus P_2 alle Produktionen $A \to Y_1...Y_k$ in P'_3 auf, die folgende Bedingungen erfüllen:

1. $Y_1...Y_k \neq \varepsilon$
2. für alle i mit $1 \leq i \leq k$:
    1. $Y_i = X_i$ für $X_i \in N_2 - N_3^{(n_2)}$
    2. $Y_i = X_i$ oder $Y_i = \varepsilon$ für $X_i \in N_3^{(n_2)}$

Wiederholen wir nun auf $\langle N_3, T_2, P'_3, S \rangle$ die Schritte 1 und 2, um eventuell neu entstandene nutzlose Symbole zu entfernen, so erhalten wir schließlich G_3.

Achtung: $\mathcal{L}(G_3) = \mathcal{L}(G) - \varepsilon$!!

Vorbereitende Schritte zur Chomsky NF: Beispiel ctd.

Schritt 3

Nun sehen wir uns die ε-Produktionen in G_2 an:

- $N_3^{(1)} = \{B\}$
- $N_3^{(2)} = \{S\} \cup \{B\}$ Also: $\varepsilon \in \mathcal{L}(G)$!

Wir erhalten somit

$$\begin{aligned} G_3 &= \langle N_3, \{a\}, P_3, S\rangle \text{ mit} \\ N_3 &= N_2 = \{S, A, B\} \\ P_3 &= \{S \to aA \mid B, A \to aB \mid a, B \to A\} \ . \end{aligned}$$

G_3 enthält nun weder nutzlose Symbole noch ε-Produktionen, und es gilt $\mathcal{L}(G_3) = \mathcal{L}(G) - \varepsilon$.

4. Elimination von Einheitsproduktionen:

Ausgehend von G_3 eliminieren wir die dort enthaltenen Einheitsproduktionen folgendermaßen:
Ist $B_0 \to B_1$ eine Produktion in P_3.
Betrachte alle Ableitungen $B_0 \Rightarrow B_1 \Rightarrow B_2 \Rightarrow ... \Rightarrow B_k \Rightarrow w, w \notin N_3$, (d.h., die im letzten Schritt angewendete Produktion ist keine Einheitsproduktion.)
Ersetze nun jede Produktion der Gestalt $B_0 \to B_1$ durch die Produktionen $B_0 \to w$.
Nachdem es jedoch passieren kann, dass durch dieses Verfahren wiederum nutzlose Symbole erzeugt werden, muss man noch einmal die Schritte 1 und 2 wiederholen, ehe man die gewünschte Grammatik G_4 erhält.

Vorbereitende Schritte zur Chomsky NF: Beispiel ctd.

Schritt 4

In G_3 gibt es noch die Einheitsproduktion $B \to A$, die wir nun eliminieren. Nachdem $B \Rightarrow A \Rightarrow aB$ bzw. $B \Rightarrow A \Rightarrow a$, ersetzen wir nun $B \to A$ durch $B \to aB \mid a$ sowie $S \to B$ durch $S \to aB \mid a$ und erhalten damit die reduzierte Grammatik

$$\begin{aligned} G_4 &= \langle N_4, \{a\}, P_4, S\rangle \text{ mit} \\ N_4 &= N_3 = N_2 = \{S, A, B\} \\ P_4 &= \{S \to aA \mid aB \mid a, A \to aB \mid a, B \to aB \mid a\} \end{aligned}$$

Wiederholt man die Schritte 1 und 2, so stellt man fest, dass in diesem Fall keine neuen nutzlosen Symbole erzeugt wurden.

Chomsky[7] Normalform

reduzierte Grammatik: Als Ergebnis der Schritte 1-4 erhalten wir eine sogenannte *reduzierte* Grammatik, also eine kontextfreie Grammatik, die weder nutzlose Symbole, noch ε- oder Einheitsproduktionen enthält.

Chomsky Normalform

Alle Produktionen besitzen die Form

$$A \to B\,C \qquad \text{oder} \qquad A \to a \qquad (a \in T,\ A, B, C \in N) \ .$$

Um das Leerwort erzeugen zu können, ist $S \to \varepsilon$ erlaubt, sofern S nicht auf der rechten Seite einer Produktion vorkommt.

Zu jeder kontextfreien Grammatik G kann man effektiv eine äquivalente Grammatik G' konstruieren, sodass $G' = \langle N', T', P', S' \rangle$ in Chomsky NF ist.

[7] Noam Chomsky, *1928

Chomsky NF (Beweis)

- Gilt $\mathcal{L}(G) = \{\}$: fertig
- Gilt $\mathcal{L}(G) \neq \{\}$: Von der reduzierten Grammatik $G_4 = \langle N_4, T_4, P_4, S\rangle$ (die $\mathcal{L}(G) - \varepsilon$ erzeugt) konstruiere G'':
  1. ersetze jedes Terminalsymbol $a \in T$ auf den rechten Seiten der Produktionen in P_4 durch eine entsprechende neue Variable X_a (soferne die rechte Seite nicht ohnehin nur aus einem einzigen Terminalsymbol besteht) und füge $X_a \to a$ hinzu.
  2. ersetze jede Produktion der Gestalt $A \to B_1 ... B_k$ für $k > 2$, durch $A \to B_1 Y_1,\ Y_1 \to B_2 Y_2,\ ...,\ Y_{k-2} \to B_{k-1} B_k$.

  Gemäß diesen Umformungen enthält G'' nur Produktionen der Gestalt $A \to a$ sowie der Gestalt $A \to BC$ für $a \in T'$ und $A, B, C \in N'$, d.h., G'' ist in Chomsky Normalform.

Chomsky NF (Beweis ctd.: $\varepsilon \in L(G)$?)

- Gilt $\varepsilon \notin L(G)$: fertig ($G' = G''$).
- Sonst:
  - S ist nicht auf der rechten Seite einer Produktion enthalten: Füge zu P' noch die Produktion $S \to \varepsilon$ hinzu ($S' = S$), um die zu G äquivalente reduzierte Grammatik G' in Chomsky Normalform zu erhalten.
  - sonst: füge noch ein neues Startsymbol S' hinzu (d.h., $N' = N'' \cup \{S'\}$) sowie die Produktionen $S' \to w$, wobei w die rechte Seite einer S-Produktion ist, d.h., $P' = P \cup \{S' \to w \mid S \to w \in P''\}$.

□

Chomsky NF - Beispiel ctd.

Schritt 5 - Chomsky NF

Wir konstruieren zu G_4 (die $\mathcal{L}(G) - \varepsilon$ erzeugt) eine reduzierte Grammatik G' in Chomsky-Normalform mit $\mathcal{L}(G') = \mathcal{L}(G)$:
Ersetze a durch X_a in allen Produktionen, in denen a nicht alleine auf der rechten Seite vorkommt:

$$\{S \to X_aA \mid X_aB \mid a, A \to X_aB \mid a, B \to X_aB \mid a, X_a \to a\}.$$

Nachdem $\varepsilon \in \mathcal{L}(G)$, muss noch $S \to \varepsilon$ hinzugefügt werden, also:

$$\begin{array}{rcl} G' & = & \langle\{S, A, B, X_a, Y_1\}, \{a\}, P', S\rangle \text{ mit} \\ P' & = & \{S \to X_aA \mid X_aB \mid a \mid \varepsilon, A \to X_aB, \\ & & A \to a, B \to X_aB \mid a, X_a \to a\} \end{array}$$

Greibach[8] Normalform für kf Grammatiken

Greibach Normalform:

Alle Produktionen haben die Form

$$A \to a\,w \qquad (a \in T,\, A \in N,\, w \in N^*)\ .$$

Erweiterte Greibach Normalform:

Alle Produktionen haben die Form

$$A \to a\,w \qquad (a \in T,\, A \in N,\, w \in (N \cup T)^*)\ .$$

Um das Leerwort erzeugen zu können, ist $S \to \varepsilon$ erlaubt, sofern S nicht auf der rechten Seite einer Produktion vorkommt.

Zu jeder kontextfreien Grammatik G kann man effektiv eine äquivalente Grammatik G' konstruieren, sodass G' in (erweiterter) Greibach NF ist.

[8] Sheila Greibach, *1939

Erweiterte Greibach Normalform - Beispiele

$L_1 = \left\{0^{2n}1^{4k}0^{2n} \mid n, k \geq 1\right\}$

Grammatik für L_1 in erweiterter Greibach Normalform:

$G_1 = \langle\{S, X\}, \{0, 1\}, P_1, S\rangle$ mit

$P_1 = \{S \to 0^2S0^2, S \to 0^2X0^2, X \to 1^4X, X \to 1^4\}$

Linksableitungen:

$S \Rightarrow^{n-1} 0^{2(n-1)}S0^{2(n-1)} \Rightarrow 0^{2n}X0^{2n} \Rightarrow^{k-1} 0^{2n}1^{4(k-1)}X0^{2n} \Rightarrow$
$0^{2n}1^{4k}0^{2n} \qquad \forall n, k \geq 1$

$L_2 = \{a^{n-4}b^ka^{5n} \mid n > 4, k \geq 3\}$

Grammatik für L_2 in erweiterter Greibach Normalform:

$G_2 = \langle\{S, A, B\}, \{a, b\}, P_2, S\rangle$ mit

$P_2 = \{S \to aAa^{25}, A \to aAa^5 \mid b^2B, B \to bB \mid b\}$

Eigenschaften kontextfreier Sprachen

Abschlusseigenschaften

Kontextfreie Sprachen sind **abgeschlossen** unter

- $\cup$, $\cdot$, *,
- Homomorphismen,
- Schnitt mit regulären Sprachen

Kontextfreie Sprachen sind **nicht abgeschlossen** unter

- Durchschnitt und Komplement.

Abschlusseigenschaften kontextfreier Sprachen

Kontextfreie Sprachen sind unter $\cup$, $\cdot$, * abgeschlossen.

Seien G_1, G_2 kontextfreie Grammatiken mit $N_1 \cap N_2 = \{\}$ und $S \notin N_1 \cup N_2$:

$$G_1 = \langle N_1, T_1, P_1, S_1 \rangle, \quad G_2 = \langle N_2, T_2, P_2, S_2 \rangle$$

Dann gilt:

$$G_{union} = \langle N_1 \cup N_2 \cup \{S\}, T_1 \cup T_2, \{S \to S_1 \mid S_2\} \cup P_1 \cup P_2, S \rangle$$
$$G_{cat} = \langle N_1 \cup N_2 \cup \{S\}, T_1 \cup T_2, \{S \to S_1 \cdot S_2\} \cup P_1 \cup P_2, S \rangle$$
$$G_{star} = \langle N_1 \cup \{S\}, T_1, \{S \to \varepsilon, S \to S_1 S\} \cup P_1, S \rangle$$

Sowie

$\mathcal{L}(G_{union})$, $\mathcal{L}(G_{cat})$, $\mathcal{L}(G_{star})$ kontextfrei

□

WKA revisited: Dyck[9] Sprachen

Dyck Sprachen

Sei D_n über $\Gamma_n = \{(_1,)_1, ..., (_n,)_n\}$ die kleinste Menge, für die gilt:

- $\varepsilon \in D_n$.
- Ist $v \in D_n$, so ist auch $(_i v)_i \in D_n, 1 \leq i \leq n$.
- Ist $v_1, v_2 \in D_n$, so ist auch $v_1 v_2 \in D_n$.

[9] Walther von Dyck, 1856-1934

Satz von Chomsky-Schützenberger[10]

Jede kontextfreie Sprache ist homomorphes Bild des Durchschnitts einer regulären Sprache mit einer Dyck-Sprache:

Satz von Chomsky-Schützenberger

Eine Sprache L über Σ ist genau dann kontextfrei, wenn zu einem $n \geq 0$ ein Homomorphismus $h : \Gamma_n^* \longrightarrow \Sigma^*$ so existiert, dass

$$L = h(D_n \cap R),$$

wobei R eine reguläre Sprache über Γ_n bezeichnet.

$L = \{a^n b^n \mid n \geq 0\}$ ist kontextfrei, da

$$L = h(D_1 \cap \{(\}^* \{)\}^*)$$

wobei $h : \{(,)\}^* \longrightarrow \{a, b\}^*$ mit $h(() = a, \quad h()) = b$

[10] Marcel-Paul Schützenberger, 1920 - 1996

Abschlusseigenschaften kontextfreier Sprachen

Kontextfreie Sprachen sind **nicht** abgeschlossen unter $\cap$.

$$L_1 = \{a^m b^m c^n \mid m, n \geq 0\} \qquad L_2 = \{a^m b^n c^n \mid m, n \geq 0\}$$

L_1, L_2 sind kontextfrei:

$$\begin{array}{lll} S_1 & \to & S_1\, c \mid A \\ A & \to & a\, A\, b \mid \varepsilon \end{array} \qquad \begin{array}{lll} S_2 & \to & a\, S_2 \mid B \\ B & \to & b\, B\, c \mid \varepsilon \end{array}$$

Nun gilt

$$L = L_1 \cap L_2 = \{a^m b^m c^m \mid m \geq 0\}$$

L ist aber **nicht** kontextfrei!

□

Abschlusseigenschaften kontextfreier Sprachen

Kontextfreie Sprachen sind nicht abgeschlossen unter Komplement

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Wir haben gesehen, dass kontextfreie Sprachen unter $\cup$ abgeschlossen sind. Wären diese auch unter Komplement abgeschlossen, so – wegen obiger Beziehung – auch unter Durchschnitt. Wir haben aber gezeigt, dass dies nicht der Fall ist.

□

Entscheidungsprobleme für kontextfreie Sprachen

Entscheidbare Probleme:

- Ist die k.f. Sprache L leer/endlich/unendlich?
- Liegt ein Wort w in der k.f. Sprache L? (Wort-Problem)

Unentscheidbare Probleme:

- Gilt $L = \Sigma^*$?
- $L_1 = L_2$ (Äquivalenzproblem)?
- $L_1 \subseteq L_2$?
- $L_1 \cap L_2 = \{\}$?
- Ist L regulär?
- Ist $L_1 \cap L_2$ bzw. $\Sigma^* - L$ kontextfrei?

Pumping Lemma für kontextfreie Sprachen

Pumping Lemma für kontextfreie Sprachen

Sei L eine unendliche kontextfreie Sprache. Dann existiert eine (nur von L abhängige) Schranke $m > 0$ so, dass für jedes Wort $w \in L$ mit $|w| \geq m$ Wörter u, v, x, y, z so existieren, dass

$$w = uvxyz \qquad \text{mit } |vxy| \leq m \text{ und } |vy| \geq 1$$

sowie

$$w_i = uv^ixy^iz \in L \qquad \text{für alle } i \geq 0.$$

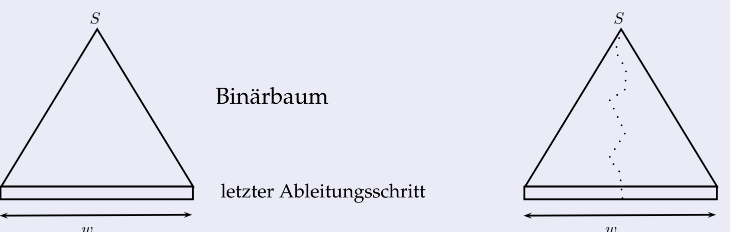
Beweis Idee:

Während man das Pumping Lemma für reguläre Sprachen mit der Anzahl der Zustände entlang eines Pfades in einem DEA begründet, kann man hier in ähnlicher Weise mit der Anzahl der Nonterminale entlang von Pfaden in den binären Ableitungsbäumen einer kontextfreien Grammatik (in Chomsky Normalform) argumentieren.

Pumping Lemma für kontextfreie Sprachen - Beweis

Beweis

Sei G eine Grammatik in Chomsky Normalform mit k Nonterminalen. Wähle $m = 2^k$. Betrachte nun den Ableitungsbaum eines beliebigen Wortes $w \in L(G)$ mit $|w| \geq m$. Dieser ist (bis auf den letzten Ableitungsschritt) ein Binärbaum:

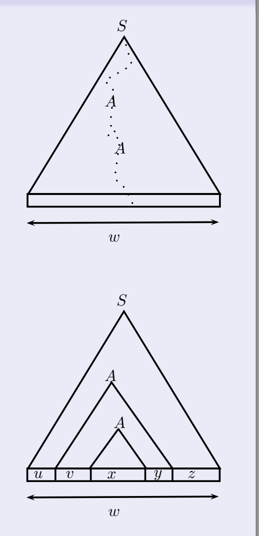


Dieser Binärbaum hat $\geq 2^k$ Blätter. Daher muss es mindestens einen Pfad der Länge $\geq k$ geben. Wir halten einen solchen Pfad fest.

Pumping Lemma für kontextfreie Sprachen - Beweis ctd.

Beweis ctd.

Einschließlich des Startsymbols befinden sich auf diesem Pfad $\geq k+1$ Nonterminale. Da die Grammatik G nur k verschiedene Nonterminale besitzt, muss mindestens eines davon doppelt vorkommen (Schubfachprinzip!).



Wir betrachten nun die Teilwörter, die aus den beiden Nonterminalen A abgeleitet werden können. Diese induzieren eine Zerlegung von w in die Teilwörter $uvxyz$.

Da das obere Nonterminal A mittels einer Produktion der Form $A \rightarrow BC$ weiter abgeleitet wird, ist (sind) v oder y (oder beide) nicht leer.

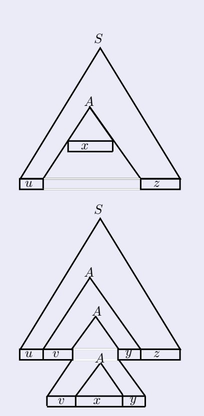
Pumping Lemma für kontextfreie Sprachen - Beweis ctd.

Beweis ctd.

Der Ableitungsbaum für w kann auf verschiedene Arten modifiziert werden:

Beispielsweise kann an das obere A der Ableitungsbaum des unteren A gehängt werden.
Wir erhalten damit eine Ableitung von $uxz = uv^0xy^0z$. Das heißt, $uv^0xy^0z \in L(G)$.

Man kann auch an das untere A den Teilbaum anhängen, der am oberen A beginnt. Dies ergibt eine Ableitung von $uvvxyyz = uv^2xy^2z$.



Allgemein gilt, dass für jedes $i \geq 0$ $uv^ixy^iz \in L(G)$. □

Pumping Lemma für k.f. Sprachen - Beispiel

$L = \{a^n b^n c^n \mid n \geq 0\}$ ist nicht kontextfrei

Beweis: indirekt. Angenommen L ist kontextfrei. Sei dann m die Konstante des Pumping Lemmas. Wähle z.B.

$$w = a^m b^m c^m$$

Für Wörter $w \in L$ gilt: $|w|_a = |w|_b = |w|_c$.

Da $|w| = 3m > m$ kann w zerlegt werden in $w = uvxyz$ mit $|vxy| \leq m$.

Daraus folgt, dass vxy nicht gleichzeitig Symbole a und Symbole c enthalten kann, d.h., entweder ist $|vxy|_a = 0$ oder $|vxy|_c = 0$.
(In der Tat liegen die Symbole a und c zu weit auseinander.)

Pumping Lemma für k.f. Sprachen - Beispiel ctd.

$$w = a^m b^m c^m, \quad w = uvxyz, \quad |vxy| \leq m, \quad |vy| \geq 1.$$

Fallunterscheidung:

Fall (a): $|vxy|_c = 0$.

Wegen $|vy| \geq 1$ gilt $|vy|_a + |vy|_b > 0$. Wähle $i = 0$, d.h.,

$$w_0 = uv^0xy^0z = uxz$$

(Nach dem Pumping Lemma müsste $w_0 \in L$ gelten.)

Für *uxz* gilt nun

$|uxz|_c = |uvxyz|_c = |w|_c$, aber $|uxz|_a < |w|_a$ oder $|uxz|_b < |w|_b$.
Das heißt, es gilt *nicht*
$|uxz|_a = |uxz|_b = |uxz|_c$ und daher $uxz \notin L$. **Widerspruch!**

Pumping Lemma für k.f. Sprachen - Beispiel ctd.

$$w = a^m b^m c^m, \quad w = uvxyz, \quad |vxy| \leq m, \quad |vy| \geq 1.$$

Fallunterscheidung ctd.:

Fall (b): $|vxy|_c > 0$ und daher $|vxy|_a = 0$.

Wegen $|vy| \geq 1$ gilt $|vy|_b + |vy|_c > 0$.

Für $w_0 = uxz$ gilt nun aber
$|uxz|_a = |uvxyz|_a = |w|_a$, jedoch $|uxz|_b < |w|_b$ oder $|uxz|_c < |w|_c$.
Das heißt, es gilt *nicht*
$|uxz|_a = |uxz|_b = |uxz|_c$ und daher $uxz \notin L$. **Widerspruch!**

Wir haben alle möglichen Zerlegungen von z untersucht, aber in jedem Fall einen Widerspruch erhalten! Somit kann L nicht kontextfrei sein. □

Korollar zum P.L. für kontextfreie Sprachen

Folgerung [Korollar zum P.L. für kontextfreie Sprachen]

Sei $L \subseteq \{a\}^*$, sodass $L = \{a^{f(n)} \mid n \geq 0\}$ für eine streng monoton wachsende Funktion f in den natürlichen Zahlen.
Gibt es für jede natürliche Zahl k eine natürliche Zahl $n(k)$, sodass

$$f(n(k+1)) - f(n(k)) \geq k,$$

dann kann L nicht kontextfrei sein.

Nach obigem Satz können die Sprachen

$$\{a^p \mid p \text{ ist eine Primzahl}\} \quad \text{und}$$
$$\{a^{2^n} \mid n \geq 0\}$$

nicht kontextfrei sein.

Kontextfreie Sprachen über einem einelementigen Alphabet

Bemerkung: Für kontextfreie Sprachen über einem einelementigen Alphabet gilt jedoch der folgende Satz:

Jede kontextfreie Sprache über einem einelementigen Alphabet ist regulär.

Jenseits der Kontextfreiheit

Formale Sprachen

$\{a^n b^n c^n \mid n \geq 0\}$ ist nicht kontextfrei.

Programmiersprachen

Typen- und Deklarationsbedingungen sind nicht oder nur schwer kontextfrei darstellbar.

Natürliche Sprachen

Schweizer Dialekt:

Jan säit das *mer d'chind em Hans es huus haend wele laa hälfe aastriiche.*

Hochdeutsch:
Jan sagt, dass *wir die Kinder dem Hans helfen lassen wollten, das Haus anzustreichen.*

Monotone Grammatiken

Monotone Grammatik

Eine Grammatik heißt *monoton*, wenn für alle Produktionen $\alpha \to \beta$ die Länge von α kleiner gleich der Länge von β ist ($\alpha \neq \varepsilon$).
Kommt das Startsymbol S auf keiner rechten Seite vor, ist auch $S \to \varepsilon$ zugelassen.

$L = \{a^n b^n c^n \mid n \geq 0\}$

$G = (\{S, T, C\}, \{a, b, c\}, P, S)$

wobei P folgende Produktionen enthält:

$$
\begin{array}{rcl}
S & \to & \varepsilon \mid abc \mid a\,T\,bc \\
T & \to & a\,T\,b\,C \mid ab\,C
\end{array}
\qquad\qquad
\begin{array}{rcl}
C\,b & \to & b\,C \\
C\,c & \to & cc
\end{array}
$$

Es gilt: $\mathcal{L}(G) = L$.

Kontextsensitive Grammatiken

Kontextsensitive Grammatik

Eine Grammatik heißt *kontextsensitiv*, wenn alle Produktionen die Form $uAv \to u\beta v$ besitzen, wobei $u, v \in (N \cup T)^*$, $A \in N$ und $\beta \in (N \cup T)^+$.
Kommt Startvariable S auf keiner rechten Seite vor, ist auch $S \to \varepsilon$ zugelassen.

Für jede Sprache, die von einer monotonen Grammatik erzeugt wird, gibt es auch eine kontextsensitive Grammatik, und umgekehrt.

Kontextsensitive Grammatiken: Beispiel

$L = \{a^n b^n c^n \mid n \geq 0\}$

$G = (\{S, T, B, C, X, Y\}, \{a, b, c\}, P, S)$

wobei P folgende Produktionen enthält:

$$
\begin{array}{rcl}
S & \to & \varepsilon \mid abc \mid aTBc \\
T & \to & aTBC \mid abC \\
B & \to & b \\
Cc & \to & cc
\end{array}
\qquad
\begin{array}{rcl}
CB & \to & CY \\
CY & \to & XY \\
XY & \to & XC \\
XC & \to & BC
\end{array}
$$

Es gilt: $\mathcal{L}(G) = L$.

Sprachfamilien

Sprachfamilien

Sei $i \in \{0, 1, 2, 3\}$ und Σ ein Alphabet. Dann wird die Menge aller formalen Sprachen $L \subseteq \Sigma^*$, die von einer Grammatik vom Typ i erzeugt werden können, mit $\mathcal{L}_i(\Sigma)$ bezeichnet. Die Familie der formalen Sprachen, die von einer Typ-i-Grammatik erzeugt werden können, bezeichnen wir mit $\mathcal{L}_i$.

Normalformen für Typ-i-Grammatiken

Typ-3:	$A \to b$,	$A \to bC$,		$b \in T$
Typ-2:	$A \to b$,	$A \to BC$,		$b \in T$
Typ-1:	$A \to b$,	$A \to BC$,	$AD \to BC$,	$b \in T$
Typ-0:	$A \to b$,	$A \to BC$,	$AD \to BC$,	$b \in T \cup \{\varepsilon\}$

Bei den Typen 3, 2, und 1 wird noch die Produktion $S \to \varepsilon$ erlaubt, sofern die Startvariable S nicht auf der rechten Seite einer Produktion vorkommt.

Wie man aus diesen Normalformen für die Typ-i Grammatiken sofort erkennt, gilt

$$\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1.$$

Rekursive Sprachen

Rekursive (entscheidbare) Sprachen

Eine Sprache $L \in \mathcal{L}_0(\Sigma)$ heißt *rekursiv* (*entscheidbar*), wenn auch das Komplement der Sprache $\overline{L} = \Sigma^* - L$, eine rekursiv aufzählbare Sprache ist, d.h., $\overline{L} \in \mathcal{L}_0(\Sigma)$. Die Menge der rekursiven Sprachen aus $\mathcal{L}_0(\Sigma)$ wird mit $\mathcal{L}_{rec}(\Sigma)$ bezeichnet, die Familie der rekursiven Sprachen mit $\mathcal{L}_{rec}$.

Wortproblem für rekursive Sprachen

Wortproblem

Das Problem $w \in L$ ist für rekursive Sprachen L entscheidbar.

Beweis. [Idee] Einfacher Entscheidungsalgorithmus:
gegeben Grammatik G für eine Sprache L sowie
Grammatik G' für das Komplement von L, d.h. $\overline{L}$
Berechne schrittweise alle möglichen Satzformen für G und G', die in $n = 1, 2, \ldots$ Schritten ableitbar sind. Da nun das Wort w entweder in $\mathcal{L}(G)$ oder $\mathcal{L}(G')$ liegt, muss es ein n geben, sodass w in n Schritten entweder in G oder in n Schritten in G' ableitbar ist, d.h., in den in n Schritten ableitbaren Satzformen vorkommt. □

Chomsky-Hierarchie

Sprachfamilie	Grammatiktyp	Maschinenmodell
$\mathcal{L}_0$ (rekursiv aufzählb.)	unbeschränkt	Turingmaschine (= EA + RAM)
$\mathcal{L}_1$ (kontextsensitiv)	kontextsensitiv monoton	Linear beschränkter Aut. (= EA + beschr.RAM)
$\mathcal{L}_2$ (kontextfrei)	kontextfrei	Kellerautomat (= EA + Stack)
$\mathcal{L}_3$ (regulär)	regulär	endlicher Automat (EA)

Chomsky Hierarchie

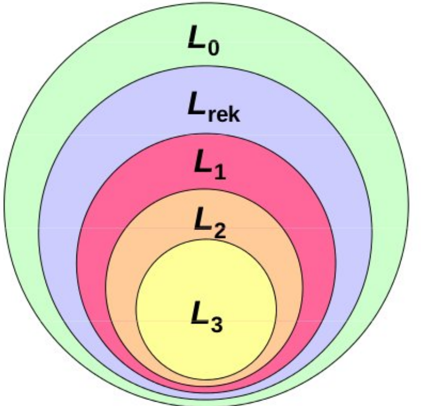
$$\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_{rec} \subsetneq \mathcal{L}_0.$$

Chomsky-Hierarchie

Chomsky Hierarchie

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_{rec} \subset \mathcal{L}_0.$$

nicht rekursiv aufzählbare Sprachen



Abschlusseigenschaften von Sprachfamilien

Abschluss unter regulären Operationen

Alle Sprachfamilien $\mathcal{L}_i$, $i \in \{0, 1, 2, 3\}$, der Chomsky-Hierarchie sind gegenüber den Operationen Vereinigung, Konkatenation und Kleene-Stern abgeschlossen.

Abschlusseigenschaften: Homomorphismen

Seien Σ und Γ zwei Alphabete. **Homomorphismus**: $h\colon \Sigma \to \Gamma^*$
Induktive Erweiterung auf Wörter über Σ:

1. $h(\varepsilon) = \varepsilon$;
2. $h(wa) = h(w)h(a)$ für alle $w \in \Sigma^*$ und $a \in \Sigma$.

Anwendung von h auf jedes Wort der Sprache L:

$$h(L) = \{h(w) \mid w \in L\}$$

Ein Homomorphismus h heißt ε-**frei**, wenn $h(a) \neq \varepsilon$ für alle $a \in \Sigma$.

Abschluss unter Homomorphismen

Die Sprachfamilien $\mathcal{L}_i$, $i \in \{0, 2, 3\}$, sind gegenüber beliebigen Homomorphismen abgeschlossen; die Familie der monotonen (kontext-sensitiven) Sprachen ist nur gegenüber ε-freien Homomorphismen abgeschlossen.

Homomorphismen: Beispiel

$L = \{a^{m^4} b^{mn} c^{mn} a^n d^{mn} e^m \mid n \geq 1\}$ nicht kontextfrei

Wir zeigen indirekt, dass L nicht kontextfrei sein kann:

Sei h der Homomorphismus mit

$$h : \{a, b, c, d, e\}^* \to \{a, b, c\}^* \text{ sowie}$$

$$h(a) = \varepsilon, \quad h(b) = a, \quad h(c) = b, \quad h(d) = c, \text{ und } h(e) = \varepsilon,$$

d.h., $h(L) = \{a^{mn} b^{mn} c^{mn} \mid n \geq 1\}$.

Die Familie der kontextfreien Sprachen ist gegenüber beliebigen Homomorphismen abgeschlossen, d.h. auch $h(L)$ müsste daher kontextfrei sein. $h(L)$ ist aber nicht kontextfrei, demnach kann auch L keine kontextfreie Sprache sein.

gsm-Abbildungen

gsm (generalized sequential machine): endlicher Automat mit Ausgabe

Eine *gsm* ist ein Sechstupel $M = \langle Q, \Sigma, \Gamma, \delta, q_0, F\rangle$, wobei

Q Zustände, Σ Eingabealphabet, Γ Ausgabealphabet,

$\delta : Q \times \Sigma \to Q \times \Gamma^*$.

q_0 Startzustand, F Endzustände

Bedeutung von $\delta(q, a) = (p, w)$: Beim Übergang von q nach p wird Symbol a gelesen und Wort w ausgegeben:

a/w

q p

gsm-Abbildungen

gsm-Abbildung $f_M : \Sigma^* \to \mathcal{P}(\Gamma^*)$
$f_M(w)$ definiert durch alle Ausgabewörter v, die sich bei Analyse des Eingabewortes w auf einem Pfad vom Startknoten q_0 zu einem Endknoten aus F ergeben.
Erweiterung auf Sprachen $L \subseteq \Sigma^*$:

$$f_M(L) := \{v \in \Gamma^* \mid v \in f_M(w) \text{ für ein } w \in L\}$$

M und f_M heißen ε-*frei*, wenn $\delta : Q \times \Sigma \to Q \times \Gamma^+$.
M und f_M heißen *deterministisch*, wenn für jedes Paar (q, a) höchstens ein $(q, a; p, v) \in \delta$.

gsm-Abbildungen: Beispiel

gsm M bildet $\{0^n10^n \mid n \geq 0\}$ auf $\{a^nb^n \mid n \geq 0\}$ ab:

$$M = \langle \{q_0, q_1\}, \{0, 1\}, \{a, b\}, \delta, q_0, \{q_1\} \rangle$$

mit
$\delta(q_0, 0) = (q_0, a)$
$\delta(q_0, 1) = (q_1, \varepsilon)$
$\delta(q_1, 0) = (q_1, b)$

0/a
0/b
1/ε
q0
q1

Homomorphismen und gsm-Abbildungen

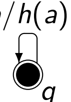
Jeder Homomorphismus ist eine sehr einfache gsm-Abbildung, die nur einen einzigen Zustand benötigt:

Für einen gegebenen Homomorphismus h ist die zugehörige gsm M_h folgendermaßen definiert:

$$M_h = (\{q\}, \Sigma, \Gamma, \delta, q, \{q\})$$

mit

$$\delta(q, a) = (q, h(a)) \quad \text{für alle } a \in \Sigma$$



Abschlusseigenschaften: gsm-Abbildungen

Abschluss unter gsm-Abbildungen

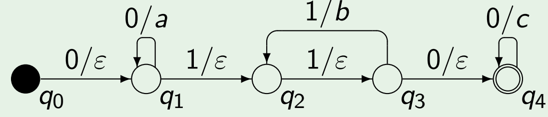
Für $i \in \{0, 2, 3\}$ sind die Familien $\mathcal{L}_i$ gegenüber beliebigen gsm-Abbildungen und damit auch gegenüber beliebigen Homomorphismen abgeschlossen; die Familie der monotonen (kontext-sensitiven) Sprachen ist nur gegenüber ε-freien gsm-Abbildungen und damit auch gegenüber ε-freien Homomorphismen abgeschlossen.

gsm-Abbildungen: Beispiel

$\{0^n 1^{2n} 0^n \mid n \geq 2\}$ nicht kontextfrei

Anmerkung: $\{a^n b^n c^n \mid n \geq 1\}$ ist bekanntermaßen nicht kontextfrei.

Beweis indirekt. Angenommen $L = \{0^n 1^{2n} 0^n \mid n \geq 2\}$ ist kf. Sei dann $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{a, b, c\}, \delta, q_0, \{q_4\})$ die (deterministische) *gsm* mit
$\delta(q_0, 0) = (q_1, \varepsilon)$, $\delta(q_1, 0) = (q_1, a)$, $\delta(q_1, 1) = (q_2, \varepsilon)$,
$\delta(q_2, 1) = (q_3, \varepsilon)$, $\delta(q_3, 1) = (q_2, b)$, $\delta(q_3, 0) = (q_4, \varepsilon)$,
$\delta(q_4, 0) = (q_4, c)$.



Da die Familie der kontextfreien Sprachen gegenüber beliebigen gsm-Abbildungen abgeschlossen ist, müsste auch $M(L) = \{a^n b^n c^n \mid n \geq 1\}$ kontextfrei sein, was aber nicht der Fall ist. Widerspruch! Somit kann auch L nicht kontextfrei sein.

Anmerkung zum vorigen Beispiel
Man beachte, dass in diesem Falle die Verwendung eines Homomorphismus nicht ausreichen würde!
Denn mittels eines Homomorphismus könnten die Symbole 0 nicht zugleich auf Symbole a und Symbole c abgebildet werden.

Abschlusseigenschaften von Sprachfamilien

	$\mathcal{L}_3$	$\mathcal{L}_2$	$\mathcal{L}_1$	$\mathcal{L}_0$
Vereinigung	ja	ja	ja	ja
Konkatenation	ja	ja	ja	ja
Kleenescher Stern	ja	ja	ja	ja
Komplement	ja	**nein**	ja	**nein**
Durchschnitt	ja	**nein**	ja	ja
Durchschnitt mit reg. Mengen	ja	ja	ja	ja
Homomorphismen	ja	ja	nein	ja
ε-freie Homomorphismen	ja	ja	ja	ja
gsm-Abbildungen	ja	ja	nein	ja
ε-freie *gsm*-Abbildungen	ja	ja	ja	ja

6.0 VU Theoretische Informatik (192.017)

Teil 2b: Berechenbarkeit (Vertiefung)

Chris Fermüller

Institut für Logic and Computation

Wintersemester 2023

Teil 2b – Berechenbarkeit (Überblick)

Teil 2b.1: Die Church-Turing These

Teil 2b.2: Imperatives Paradigma: Turingmaschinen und mehr

Teil 2b.3: Funktionales Paradigma: λ-Kalkül

Teil 2b.4: Rekursive Funktionen

Teil 2b.5: Der Satz von Rice und seine Anwendung

Teil 2b.6: Weitere unentscheidbare Probleme

Teil 2b.7: Unkonventionelle Berechenbarkeitsmodelle

Folgendes wissen wir aus Teil 1:

- Es gibt konkrete unentscheidbare (Entscheidungs-)Probleme: z.B. das Halteproblem (und viele mehr)
- Unentscheidbare Probleme sind oft semi-entscheidbar: Halteproblem, Gültigkeit prädikatenlogischer Formeln, Korrektheit bezogen auf einzelne Input/Output-Paare, Code-Erreichbarkeit, ...
- Viele wichtige Probleme sind nicht einmal semi-entscheidbar: z.B. Programmäquivalenz, universelles Halteproblem, Korrektheit bezogen auf allgemeine Vor- und Nach-Bedingungen ($\rightarrow$ Teil 4), ...
- Ähnliches gilt für Funktionen: Die meisten (überabzähl viele) Funktionen sind unberechenbar ($\rightarrow$ Diagonalverfahren)

Folgendes wissen wir aus Teil 2a:

- Eingeschränkte Rechenmodelle bzw. Darstellungsmethoden führen zur Chomsky-Hierarchie von Sprachen, also (Entscheidungs-)Problemen
- Chomsky-Typ 0 = semi-entscheidbar = rekursiv aufzählbar
- Turingmaschinen (TMs) eignen sich zur mathematischen Modellierung

Wie robust ist unser Berechenbarkeitsmodell?

Beobachtung: “Entscheidbarkeit” und “Berechenbarkeit” wurde in Teil 1 – über die Sprache SIMPLE – flexibel, aber etwas informell – definiert.
Das ist für viele Zwecke adäquat, lässt aber folgende Frage offen:

- Führen unterschiedliche Modelle zu unterschiedlichen Begriffen von Berechenbarkeit? Oder sind “entscheidbares Problem” und “berechenbare Funktion” unabhängig vom Berechenbarkeitsmodell?

Church-Turing-These (Variante 1 – Turing)

Wenn eine Funktion in irgendeinem konkreten Sinn berechenbar ist, so ist sie auch schon auf einer Turingmaschine berechenbar.

Church-Turing-These (Variante 2 – Turing)

Gibt es ein endlich beschreibbares Verfahren zur exakten Spezifizierung einer formalen Sprache L, so gibt es eine Turingmaschine, die L akzeptiert.

Fragen zur Church-Turing-These

- Wie kommt es zu dieser These?
- Wer war Church und was hat er damit zu tun?
- Wie haben Turing bzw. Church für ihre These argumentiert?
- Wie hat die Fachwelt reagiert?
- Gibt es ernsthafte Versuche die These zu ‘beweisen’ oder widerlegen?
- Welche Bedeutung hat die These für die Informatik?

Eine wichtige Beobachtung:

Formale Sprachen und Funktionen lassen sich ineinander übersetzen:
– Sprache ($\subseteq \Sigma^*$) $\longrightarrow$ charakteristische Funktion $\Sigma^* \to \{0, 1\}$
– Funktionen $\mathbb{N}^k \to \mathbb{N}^m$ oder $(\Sigma^*)^k \to (\Sigma^*)^m$ sind Mengen von Paaren von Tupeln, daher darstellbar als Mengen von Wörtern, also als Sprachen

Auch die Beschränkung auf $\mathbb{N}$ bzw. auf Strings aus $\{0, 1\}^*$ ist für die prinzipielle Berechen- oder Entscheidbarkeit unwesentlich.

Ein Problem von David Hilbert

"Wir müssen wissen, und wir werden wissen."

- David Hilbert (1862-1943)
- einer der größten und produktivsten Mathematiker aller Zeiten
- "... Eine Diophantische Gleichung [...] sei vorgelegt: man soll ein Verfahren angeben, nach welchem sich mittelst einer endlichen Anzahl von Operationen entscheiden lässt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist ..."

Ähnlich: Entscheidungsproblem für prädikatenlogische Gültigkeit

Hilbertsches Programm: Formalisierung der gesamten Mathematik um ihre Widerspruchsfreiheit beweisen zu können

David Hilbert: Mathematische Probleme - Vortrag, gehalten auf dem internationalen Mathematiker-Kongreß zu Paris 1900. In: Nachrichten von der Königl. Gesellschaft der Wissenschaften zu Göttingen. Mathematisch-Physikal. Klasse, Heft 3, 1900, 253-297

Was motivierte Church und Turing?

Antwort: Eine negative Lösung zu Hilberts Problemen zu finden!

Beobachtung zu positiven und negativen Instanzen von ‘Verfahren’:
Positiv: ein informelles Modell der Berechenbarkeit genügt.
Negativ (=Unberechenbarkeitsbeweis): verlangt ein präzises Modell!

Angeregt durch Hilberts Entscheidungsproblem schlug Church 1936 eine mathematische Definition von Berechenbarkeit vor ($\rightarrow$ 2b.3: λ-Kalkül).
Problem: nicht intuitiv, (zumindest zunächst:) undurchsichtig.

Turings Antwort: Analysiere zuerst was ein menschlicher Rechner tut!
Unterscheide dabei unwesentliche und prinzipielle Beschränkungen.

Beobachtung: Nur prinzipielle Beschränkungen sind zunächst für eine brauchbare Definition von Berechenbarkeit relevant. Spezifische (Zeit-,Platz-,Mittel-)Beschränkungen können danach ‘eingebaut’ werden.
($\rightarrow$ Chomsky-Hierarchie (Teil 2a), $\rightarrow$ Komplexitätstheorie (Teil 3))

Erst Turings Analyse (siehe 2b.2) hat die Fachwelt überzeugt (Gödel, ...)!

Gibt es ernsthafte Beweis- oder Widerlegungs-Versuche?

Antwort: erstaunlich(?) wenige!
Diagnose: Turing war wohl sehr überzeugend.
Außerdem habe sich dutzende andere Modelle als äquivalent erwiesen.

Eine interessante Alternative zu Turingmaschinen:
Kolmogorov(-Uspenski)-Maschinen beruhen ebenfalls auf einer grundlegenden Analyse von 'Rechenverfahren' und beschreiben eine Variante von sogenannten Pointer-Maschinen, die nicht auf einem Band, sondern auf bestimmten Graphen-Strukturen arbeiten.
Siehe `https://esolangs.org/wiki/Kolmogorov_machine`

Aus informatischer Sicht wichtiger: Registermaschinen.

Vorschau:
Wir werden im Folgenden nicht nur die Modelle von Turing und Church präsentieren, sondern auch Registermaschinen, sowie einen weiteren wichtigen mathematischen Zugang zu Berechenbarkeit präsentieren.

Relevanz für die (heutige) Informatik

Zwei Aspekte:

(1) Präzise und robuste Modelle der Berechenbarkeit sind eine Voraussetzung für 'Unmöglichkeitsbeweise'.
Solche Beweise sind oft durchaus praktisch relevant, weil Sie aufzeigen, wenn, warum und wie bestimmte Einschränkungen von wichtigen Aufgabenstellungen angebracht sind. Das betrifft, z.B., Korrektheitsbeweise, Redundanzerkennung (optimale Compiler,..), Äquivalenz von Spezikationen, Äquivalenz von Programmen, etc.

(2) Die sehr unterschiedlichen Zugänge von Church und Turing verweisen auf unterschiedliche Paradigmen von Programmiersprachen:
- TM $\rightarrow$ SIMPLE $\rightarrow$ Java, C, C++,...: imperatives Paradigma
- λ-Kalkül $\rightarrow$ ML, Haskell, ...: funktionales (deklaratives) Paradigma

Teil 2b – Berechenbarkeit (Überblick)

Teil 2b.1: Die Church-Turing These

Teil 2b.2: Imperatives Paradigma: Turingmaschinen und mehr

Teil 2b.3: Funktionales Paradigma: λ-Kalkül

Teil 2b.4: Rekursive Funktionen

Teil 2b.5: Der Satz von Rice und seine Anwendung

Teil 2b.6: Weitere unentscheidbare Probleme

Teil 2b.7: Unkonventionelle Berechenbarkeitsmodelle

Turings Analyse eines (menschlichen!) Rechners

Turings Frage: Welche reelle Zahlen sind (intuitiv) berechenbar?
(Impliziert auch Antworten für Funktionen und Entscheidungsprobleme!)

Auszüge aus Alan Turing, 'On computable numbers, with an application to the Entscheidungsproblem' (1936):

The arguments which I shall use are of three kinds.
(a) A direct appeal to intuition.
(b) A proof of the equivalence of two definitions. [...]
(c) Giving examples of large classes of numbers which are computable.

Computing is normally done by writing certain symbols on paper.

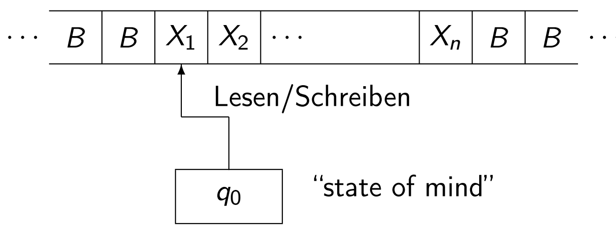
... two-dimensional character of the paper [is] always avoidable ...

... restriction to [finite] number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. ...

The behaviour of the computer at any moment is determined by the symbols which [s]he is observing, and his "state of mind" at that moment...

We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind...

Resultat der Analyse: Turingmaschinen [WH]



- Das Band enthält die Eingabe. Alle anderen Zellen enthalten ein Blank-Symbol B. (Nur endliche viele Zellen sind nicht B-gefüllt.)
- Lese/Schreib-Kopf ist immer über einer Bandzelle positioniert.
- Rechenschritt: Ersetze das gelesene Symbol (optional); schiebe den Lese/Schreib-Kopf (möglicherweise) eine Zelle nach links oder rechts; endet in einem (möglicherweise) neuen Zustand ("state of mind").
- Anfang: Kopf ist über der am weitesten links stehenden Eingabezelle.

Achtung: Im Unterschied zu Teil 2a gibt es nun kein Eingabeband mehr! Außerdem ist das Band nach beiden Seiten offen.

Formale Beschreibung einer Turingmaschine **[WH]**

Eine Turingmaschine besteht aus 7 Komponenten:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F),$$

- Q: endliche Menge von Zuständen,
- Σ: endliche Menge der Eingabesymbole,
- Γ: endliche Menge der Bandsymbole, $\Sigma \subsetneq \Gamma$,
- Übergangsfunktion $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R, S\}$,
- Startzustand $q_0 \in Q$,
- Blank-Symbol: $B \in \Gamma - \Sigma$,
- Endzustände: $F \subseteq Q$.

Übergänge (= Rechenschritte) **[WH]**

$$(q, X; p, Y, D) \in \delta$$

M liest im Zustand q auf dem Band Symbol X

- wechselt nun davon abhängig in den Zustand p,
- überschreibt Symbol X durch Symbol Y,
- bewegt den Lese/Schreib-Kopf wie durch D beschrieben:
  L... nach links, R... nach rechts, bzw. S... gar nicht.

Wir betrachten im Folgenden nur deterministische Maschinen:

Deterministische Turingmaschine

Eine Turingmaschine M heißt deterministisch, wenn für alle $(q, X) \in Q \times \Gamma$ höchstens ein Element $(q, X; p, Y, D) \in \delta$ existiert, $D \in \{L, R, S\}$; wir schreiben dann auch $\delta(q, X) = (p, Y, D)$.

M hält/terminiert (in q) wenn $q \in F$ ($\delta(q, X)$ bleibt dann undefiniert.)

Konfiguration / Übergang / Berechnung [WH]

Konfiguration: $X_1X_2 \dots X_{i-1}qX_iX_{i+1} \dots X_n$

TM ist in Zustand q, der Lese/Schreib-Kopf steht über X_i

links von X_1 und rechts von X_n stehen nur Bs

$\Rightarrow$ ... Übergang zwischen Konfigurationen in einem Schritt

$\Rightarrow^*$ ... Übergang zwischen Konfigurationen in beliebig vielen Schritten

- Bewegung nach links, d.h. $\delta(q, X_i) = (p, Y, L)$:

$$X_1X_2 \dots X_{i-1}\boldsymbol{qX_i}X_{i+1} \dots X_n \Rightarrow X_1X_2 \dots X_{i-2}\boldsymbol{p}X_{i-1}\boldsymbol{Y}X_{i+1} \dots X_n$$

- Bewegung nach rechts, d.h. $\delta(q, X_i) = (p, Y, R)$:

$$X_1X_2 \dots X_{i-1}\boldsymbol{qX_i}X_{i+1} \dots X_n \Rightarrow X_1X_2 \dots X_{i-1}\boldsymbol{Yp}X_{i+1} \dots X_n$$

- Lese/Schreib-Kopf bleibt stehen, d.h., $\delta(q, X_i) = (p, Y, S)$:

$$X_1X_2 \dots X_{i-1}\boldsymbol{qX_i}X_{i+1} \dots X_n \Rightarrow X_1X_2 \dots X_{i-1}\boldsymbol{pY}X_{i+1} \dots X_n$$

Akzeptierte Sprache / rekursiv aufzählbar / rekursiv **[WH]**

Akzeptierte Sprache

Die von der Turingmaschine* M akzeptierte Sprache $L(M)$ besteht aus jenen Wörtern, bei deren Eingabe M terminiert:

$$L(M) = \{w \in \Sigma^* \mid q_0 w \Rightarrow^* \alpha p \beta, \quad p \in F, \quad \alpha, \beta \in \Gamma^*\}$$

Rekursiv aufzählbare (oder: semi-entscheidbar) Sprachen

L heißt rekursiv aufzählbar wenn sie von einer TM akzeptiert wird.

Rekursive (oder: entscheidbare) Sprachen

L heißt rekursiv wenn sie von einer TM akzeptiert wird, die immer hält.

Anmerkung: Die Bezeichnungen semi-entscheidbar und entscheidbar beziehen sich auf die Variante 2 der Church-Turing-These.

*zur Erinnerung: Wir betrachten hier nur deterministische TMs!

Berechnung arithmetischer Funktionen ($\mathbb{N}^k \to \mathbb{N}$)

$[n]_2$ ... Binärdarstellung von $n \in \mathbb{N}$

Anfangskonfiguration einer TM M mit $\Sigma = \{0, 1\}$ für $(n_1, ..., n_k)$:
$q_0[n_1]_2 B...B[n_k]_2$ ... 'M arbeitet auf Eingabe $(n_1, ..., n_k)$'

Endkonfiguration einer (Funktionen-berechnenden) TM M:
$xp[n]_2 By$, $p \in F$, $x, y \in \Gamma^*, n \in \mathbb{N}$ ... 'M gibt n aus'

Berechnung von $f : \mathbb{N}^k \to \mathbb{N}$

Eine TM M mit $\Sigma = \{0, 1\}$ berechnet $f : \mathbb{N}^k \to \mathbb{N}$ wenn M auf Eingabe $(n_1, ..., n_k)$ arbeitend, $f(n_1, ..., n_k)$ ausgibt. Schreibweise: $F(M) = f$.

Anmerkung: M terminiert nicht, wenn $f(n_1, ..., n_k)$ undefiniert ist.

Turing-Berechenbarkeit einer arithmetischen Funktion

$f : \mathbb{N}^k \to \mathbb{N}$ heißt Turing-berechenbar, wenn $F(M) = f$ für eine TM M.

$\to$ Church-Turing-These (Variante 1): Berechenbar = Turing-berechenbar

Beispiel

$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_2\})$,

δ gegeben durch:

Zustand	0	1	B
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	$(q_1, 0, S)$
q_1	$(q_1, 0, L)$	$(q_1, 1, L)$	(q_2, B, R)

Beispielrechnung:

Eingabe: 6, daher auf dem Band $[6]_2 = 110$

$$\begin{array}{cccccccccc} & q_0 110 & \Rightarrow & 1q_0 10 & \Rightarrow & 11q_0 0 & \Rightarrow & 110q_0[B] & \Rightarrow 110q_1 0 \\ \Rightarrow & 11q_1 00 & \Rightarrow & 1q_1 100 & \Rightarrow & q_1 1100 & \Rightarrow & q_1 B1100 & \Rightarrow q_2 1100 \end{array}$$

Ausgabe: 12, weil auf dem Band $1100 = [12]_2$

Berechnete Funktion: $f_1 = F(M) : \mathbb{N} \to \mathbb{N}$, wobei $f_1(n) = 2n$

Achtung: Dieselbe TM M, angewendet auf Zahlenpaare, berechnet
$f_2 = F(M) : \mathbb{N}^2 \to \mathbb{N}$, wobei $f_2(m, n) = m2^{bl(n)+1} + n$,
dabei ist bl die binäre Länge: $bl(k+1) = \lfloor \log_2(k+1) \rfloor + 1$, $bl(0) = 1$

Universelle Turingmaschinen

Beobachtung:
Eine TM entspricht nicht einem Computer, sondern einem Programm!

Um ein Modell eines Rechners/Computers zu erhalten, benötigen wir eine Maschine, die beliebige Programme (=TMs) simulieren kann.

TM als Input: $\langle M \rangle$ bezeichnet den (Zahlen-)Code einer Turingmaschine M.

Universelle Turingmaschine

Eine TM U heißt universell, wenn für jede TM M und alle $(n_1, \ldots, n_k) \in \mathbb{N}^k$ gilt: $F(U)(\langle M \rangle, n_1, \ldots, n_k) = F(M)(n_1, \ldots, n_k)$.

Wichtig: Die in dieser Formulierung gewählte Einschränkung auf Funktionen-berechnende TMs ist praktisch, aber unwesentlich!

Theorem (Turing, 1936)

Es gibt universelle Turingmaschinen.

Beobachtung: Jede universelle TM entspricht einem Interpreter.

Varianten von Turingmaschinen

Die folgenden Varianten von Turings Modell sind jeweils berechnungsäquivalent, d.h. es werden die selben arithmetischen Funktionen berechnet bzw. die selben Sprachen akzeptiert:

TM-Varianten

- (Wie schon in Teil 2a): getrenntes Eingabeband, einseitiges Arbeitsband; auch ein getrenntes Ausgabeband wäre möglich
- Mehrband-TM: mehere (Arbeits-, Eingabe-, Ausgabe-)Bänder, jedes mit einem eigenen Lese/Schreib-Kopf
- Mehrkopf-Bänder: Mehrere Lese/Schreib-Köpfe pro Band
- Binäres oder unäres Bandalphabet ($\Gamma = \{0, 1\}$ oder $\Gamma = \{1\}$): Berechnungsäquivalenz modulo entsprechender Codierung
- Nichtdeterministische TM: Jede TM lässt sich 'determinisieren'!

Äquivalenzbeweise jeweils durch Reduktion (Simulation)!

Ein anderes imperatives Modell: Registermaschinen

Motivation: Wir suchen ein Modell, dass einerseits ähnlich einfach wie TMs, anderseits aber näher an heutigen Computern ist. Wir fokussieren dabei auf arithmetische Funktionen (Typ $\mathbb{N}^k \to \mathbb{N}$).

Definition einer Registermaschine (RM)

Eine m-Registermaschine (m-RM) besteht aus 3 Komponenten:

$$R = (S, \boldsymbol{O}, \boldsymbol{T})$$

- $S = \mathbb{N}^m$: Speicherzustände (= Inhalt der m Register)
- $\boldsymbol{O} = \{A_i \mid 1 \leq i \leq m\} \cup \{S_i \mid 1 \leq i \leq m\}$: Speicherbefehle
- $\boldsymbol{T} = \{t_i \mid 1 \leq i \leq m\}$: Testbefehle

Wichtig: Anders als bei TMs werden bei RMs Maschinen und Programme getrennt. Erst mit einem RM-Programm kann man rechnen!

Programme für Registermaschinen

Definition eines Registermaschinen-Programms

Ein (m)-RM-Programm besteht aus einer Startmarke $a \in \mathbb{N}$ und einer endlichen Menge aus Anweisungen:

- Funktionsanweisung: (r, f, p) lies 'r: do f then goto p'
- Testanweisung (r, t, p, q) lies 'r: if t then goto p else goto q'

r heißt Kennmarke der Anweisung, p bzw. q Sprungmarken $(r, p, q \in \mathbb{N})$.
Sprungmarken, die keine Kennmarken sind, heißen Endmarken.

$R(i)$ ... Inhalt des i-ten Registers $(R(i) \in \mathbb{N})$

Semantik der Befehle:

- A_i: $R(i) := R(i) + 1$
- S_i: $R(i) := R(i) \dot{-} 1$ $[R(i)$ bleibt 0, falls $R(i) = 0]$
- t_i: Ist $R(i) = 0$?

Wir betrachten hier nur deterministische Programme:
Jede Kennmarke kommt nur einmal als solche vor.

Rechnen mit RM-Programmen

Berechnung einer Funktion $f : \mathbb{N}^k \to \mathbb{N}$ $(k \leq m)$:
Anfangszustand: $R(i) = n_i$ für $1 \leq i \leq k$, $R(i) = 0$ für $k+1 \leq i \leq m$
Endzustand: Endmarke erreicht, $f(n_1, ..., n_k) = R(1)$

Konfigurationen:
p:$(n_1, \ldots, n_m)$ .... aktuelle Kennmarke ist p, $R(i) = n_i$ für $1 \leq i \leq m$

Ein Additions-Programm:
$ADD = (0,\ \{(0, t_2, 5, 1),\ (1, S_2, 2),\ (2, A_1, 0)\})$

Beispielrechnung:

$$\begin{array}{rl} & 0{:}(5,2) \Rightarrow 1{:}(5,2) \Rightarrow 2{:}(5,1) \Rightarrow 0{:}(6,1) \\ \Rightarrow & 1{:}(6,1) \Rightarrow 2{:}(6,0) \Rightarrow 0{:}(7,0) \Rightarrow 5{:}(7,0) \end{array}$$

Ein Substraktions-Programm ($\dot{-}$):
$SUB = (0,\ \{(0, t_2, 5, 1),\ (1, S_2, 2),\ (2, S_1, 0)\})$

Beispielrechnung:

$$\begin{array}{rl} & 0{:}(5,2) \Rightarrow 1{:}(5,2) \Rightarrow 2{:}(5,1) \Rightarrow 0{:}(4,1) \\ \Rightarrow & 1{:}(4,1) \Rightarrow 2{:}(4,0) \Rightarrow 0{:}(3,0) \Rightarrow 5{:}(3,0) \end{array}$$

Partielle Funktionen / Codierung

Programm für die (echte) Vorgänger-Funktion:
$PRED = (0,\ \{(0, t_1, 0, 1),\ (1, S_1, 2)\})$

Beispielrechnungen:

$$0{:}(2) \Rightarrow 1{:}(2) \Rightarrow 2{:}(1)$$

bzw. $0{:}(0) \Rightarrow 0{:}(0) \Rightarrow 0{:}(0) \Rightarrow$ ... (terminiert nicht)

Anmerkung: Wie TMs, so können auch RM-Programme auf (Codes von) RM-Programmen (z.B. auch auf sich selbst) angewendet werden.

$\langle P \rangle \in \mathbb{N}$ ... Zahlencode des RM-Programs P

Hinweise zur Codierung als Zahlen:
Basis vieler Codierungen ist die Eindeutig der Primfaktorzerlegung:
$\langle n_1, \ldots, n_m \rangle \rightarrow 2^{n_1} \cdot 3^{n_2} \cdot \ldots \cdot p_m^{n_m}$ ist eindeutig decodierbar!
Auch beliebige Strings über einem Alphabet Σ können so eindeutig als einzelne Zahlen codiert werden, indem man den Zeichen des Alphabets jeweils eine bestimmte Zahl zuordnet.

Wichtige Fakten zu Registermaschinen

- Es gibt universelle RM-Programme P_U: Sei $\langle F \rangle$ der Code eines RM-Programms, das $f : \mathbb{N}^k \to \mathbb{N}$ dann berechnet, dann berechnet P_U die Funktion $u : \mathbb{N}^{k+1} \to \mathbb{N}$, sodass $u(\langle F \rangle, n_1, ..., n_k) = f(n_1, ..., n_k)$.
- RMs und TMs sind berechungsäquivalent. Das stützt:

Church-Turing-These (Variante 3 – Shepherdson & Sturgis)

Wenn eine arithmetische Funktion informell berechenbar ist, so ist sie auch schon auf einer Registermaschine berechenbar.

- Eine erstaunliche Tatsache:
  Bereits 2-RMs sind – modulo geeigneter Codierungen von Inputs und Outputs als einzele Zahlen – berechnungsuniversell!

Übliche (höhere) Programmiersprachen

Frage:

Wie sieht man ein, dass SIMPLE bzw. auch übliche Programmiersprachen, implementiert auf diversen Computerarchitekturen (im Prinzip) nicht mehr berechnen können als TMs oder RMs?

Antwort:

Durch die Tatsache, dass alle Programmiersprachen in ein jeweilige Maschinensprache übersetzbar sein müssen. Auf Maschinen-Ebene lassen sich dann (oft aufwändig) geeignete Reduktionen/Simulationen finden.

(Umgekehrt kann man TMs und RMs in höheren Sprachen simulieren.)

Anmerkungen:

- Jeder Einzelfall unterstützt die Church/Turing-These
- Universelle Maschinen entsprechen Interpretern bzw. Compilern

Beobachtung: Alle bisher betrachten Modelle/Sprache sind imperativ!

Bereits Church hatte ein grundlegend anderes Modell im Sinn...

Teil 2b – Berechenbarkeit (Überblick)

Teil 2b.1: Die Church-Turing These

Teil 2b.2: Imperatives Paradigma: Turingmaschinen und mehr

Teil 2b.3: Funktionales Paradigma: λ-Kalkül

Teil 2b.4: Rekursive Funktionen

Teil 2b.5: Der Satz von Rice und seine Anwendung

Teil 2b.6: Weitere unentscheidbare Probleme

Teil 2b.7: Unkonventionelle Berechenbarkeitsmodelle

Alonzo Church – Ein Mann mit einer großen Idee ...

Herausforderung durch Hilberts Entscheidungsproblem:

> *"... propose a definition of effective calculability which [...] correspond[s] satisfactorily to the somewhat vague intuitive notion"*

- Alonzo Church (1903 - 1995)
- US-amerikanischer Mathematiker, Logiker und Philosoph und einer der Begründer der theoretischen Informatik
- Entwicklung des Lambda-Kalküls
- Zahlreiche berühmte Dissertanten: Alan Turing, Michael Rabin, Hartley Rogers, J. Barkley Rosser, Dana Scott, Raymond Smullyan, Stephen C. Kleene, ...

A. Church: An Unsolvable Problem of Elementary Number Theory, American Journal of Mathematics 58 (2), 1936, 345 – 363.

Church's große Idee: Alles ist eine Funktion!

Nicht nur Rechenvorschriften (Programme), sondern auch Zahlen, Wahrheitswerte, Listen, etc., lassen sich als Funktionsausdrücke darstellen!

Ein Notationsproblem:
'$m + n$' ist zweideutig:
Ist die Zahl $m + n$ gemeint? Oder die (2-stellige) Additions-Funktion?
Mit anderen Worten: Ist die Funktion oder deren Ergebnis gemeint?

Die Lösung: λ-Notation ('Lambda-Notation')
$\lambda m \lambda n.m + n$ bezeichnet die Additions-Funktion als solche.
Entsprechend bezeichnet, z.B., $\lambda m.m + n$ die 1-stellige Funktion '$+n$'.

Anstatt:

> *"Es sei f die Funktion $x \mapsto x + x^3$. Es sei weiters $a = f(5)$."*

können wir nun schreiben:

> "$a = (\lambda x.x + x^3)(5)$"

Die kompakteste Programmiersprache der Welt: λ-Terme

Definition von λ-Termen

- Jede Variable ist ein λ-Term $(x, y, z, \ldots, f, g, \ldots, p, q \ldots)$.
- Wenn M und N λ-Terme sind, dann ist (MN) ein λ-Term; genannt die Anwendung (oder Applikation) von M auf N.
- Wenn x eine Variable und M ein λ-Term ist, dann ist $(\lambda x.M)$ ein λ-Term. M nennt man den Body der Abstraktion $(\lambda x.M)$.

In Backus-Naur Form:

<lterm> ::= <var> | (<lterm><lterm>) | (λ <var>.<lterm>)

<var> ::= $x \mid y \mid z \mid \ldots \mid f \mid g \mid \ldots \mid p \mid q \mid \ldots$

Schreibkonventionen:

- Äußere Klammern können weggelassen werden.
- Links-Assoziativität: MNL bedeuted $(MN)L$.
- Der Body einer Abstraktion reicht so weit nach rechts wie möglich: Z.B. steht $\lambda x.MN$ für $\lambda x.(MN)$ – und nicht für $(\lambda x.M)N$.
- Wir schreiben $\lambda xyz.M$ statt $\lambda x.\lambda y.\lambda z.M$ (etc.)

Beispiele von λ-Termen (ohne Schreibkonventionen)

(xy), $(x(yz))$, $((xy)z)$, $(\lambda x.x)$, $(\lambda x.y)$, $((\lambda x.x)y)$, $((\lambda x.x)(\lambda x.x))$,
$\lambda x.(\lambda y.(\lambda z.((fx)z)))$, $(\lambda x.(\lambda y.(\lambda z.((fx)z)))(f(\lambda x.(\lambda f.(fx)))))$

Dieselben Beispiele mit Schreibkonventionen

xy, $x(yz)$, xyz, $\lambda x.x$, $\lambda x.y$, $(\lambda x.x)y$, $(\lambda x.x)\lambda x.x$,
$\lambda xyz.fxz$, $(\lambda x.\lambda y.\lambda z.fxz)(f\lambda xf.fx)$

Freie Variablen (FV) und gebundene Variablen (GV)

Definition von $FV(t)$ und $GV(t)$

$FV(x) = \{x\}$
$FV(MN) = FV(M) \cup FV(N)$
$FV(\lambda x.M) = FV(M) - \{x\}$

$GV(x) = \{\}$
$GV(MN) = GV(M) \cup GV(N)$
$GV(\lambda x.M) = GV(M) \cup \{x\}$

Beispiele:
$FV(\lambda x.xy) = \{y\}$, $GV(\lambda x.xy) = \{x\}$,
$FV(\lambda yx.x)x = \{x\}$, $GV(\lambda yx.x)x = \{x, y\}$

λ-Terme in denen alle Variablen gebunden sind heißen Kombinatoren

Currying – Die Würze des funktionalen Programmierens:

- Haskell B. Curry (1900-1982)
- US-amerikanischer Logiker und Mathematiker
- nach ihm ist die funktionale Programmiersprache HASKELL benannt.

Currying stellt mehrstellige Funktion als iterative Anwendung 1-stelliger da:
$f : A_1 \times A_2 \times \ldots \times A_n \to B$ wird zu $f_{curried} : A_1 \to (A_2 \to (\ldots A_n \to B)\ldots)$
Beachte: verlangt Prefix-Notation: $m + n \to +(m, n) \to (+_{curried}\, m)n$

Beobachtung: λ-Terme machen konsequent von Currying Gebrauch

Currying wird nicht nur in HASKELL, sondern in praktisch allen gebräuchlichen Programmiersprachen verwendet:
JavaScript, Java, C++, C#, C, Python, Scheme, Ruby, ...

Currying hat auch Anwendungen in der Linguistik
siehe `https://de.wikipedia.org/wiki/Currying`

Auswerten durch Einsetzen: β-Reduktion

λ-Terme werden durch sukzessive Substitution (=Einsetzen) des Argument-Terms in durch λ gekennzeichnete Variablen ausgewertet

Umbenennung gebundener Variablen (wie bei Quantoren): $\lambda x.x = \lambda y.y$
$M\{y/x\}$ steht für 'gebundene Vorkommen von x in M durch y ersetzt'†

Substitution der freien Vorkommen von x durch N: $M[N/x]$

$x[N/x] = N$
$y[N/x] = y$, wenn $x \neq y$
$(MP)[N/x] = (M[N/x])(P[N/x])$

$(\lambda x.M)[N/x] = \lambda x.M$
$(\lambda y.M)[N/x] = \lambda y.(M[N/x])$, wenn $x \neq y$, $y \notin FV(N)$
$(\lambda y.M)[N/x] = \lambda y'.(M\{y'/y\}[N/x])$, wenn $x \neq y$, $y \in FV(N)$, y' neu

β-Reduktion

Ein (Teil-)Term der Form $(\lambda x.M)N$ in t wird in t durch $M[N/x]$ ersetzt ($= t'$). Man schreibt $t \to_\beta t'$ und nennt $(\lambda x.M)N$ einen Redex in t.
Ein λ-Term ohne Redex ist in Normalform.

†Variablen-Umbenennung heißt auch α-Reduktion: $M \to_\alpha M\{y/x\}$

λ-Terme = Programme / Berechnen = β-Reduzieren

Beispiel:

$$\begin{aligned}(\lambda x.y)(\underline{(\lambda z.zz)(\lambda w.w)}) &\rightarrow_\beta (\lambda x.y)(\underline{(\lambda w.w)(\lambda w.w)}) \\ &\rightarrow_\beta \underline{(\lambda x.y)(\lambda w.w)} \\ &\rightarrow_\beta y\end{aligned}$$

Mit y ist natürlich eine Normalform erreicht.

Auch eine andere Redex-Wahl ist möglich:

$$\underline{(\lambda x.y)((\lambda z.zz)(\lambda w.w))} \rightarrow_\beta y$$

Beobachtungen:

- Reduktion kann neue Redexe erzeugen
- Durch Reduktion können Redex verschwinden
- Anzahl der Schritte hängt von der Reduktions-Strategie ab

Rechenergebnis = Normalform

2 wichtige Fragen:

- (1) Gibt es zu jedem Term eine Normalform?
- (2) Ist die Normalform eines Terms eindeutig?

Antwort auf Frage 1: nein!

$$(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta \ldots$$

Wie zu erwarten war: Nicht jede Berechnung terminiert!

Antwort auf Frage 2: ja!

Theorem (Church-Rosser — starke Normalisierung)

Normalformen sind eindeutig: Hat ein λ-Term eine Normalform, so führt jede Reihenfolge von β-Reduktion auf das selbe Ergebnis (Normalform).

Programmieren im λ-Kalkül (1)

Der (nicht 'das') Lambda-Kalkül formalisiert Reduktion als Regelsystem.

Boolesche Funktionen:
Wir definieren Wahrheitswerte und Wahrheitsfunktionen als Kombinatoren:

Booleans / Konditional

$$\begin{aligned} \mathbf{T} &= \lambda xy.x \\ \mathbf{F} &= \lambda xy.y \\ \mathbf{if_then_else} &= \lambda pxy.pxy \end{aligned}$$

Beispiel:

$$\begin{aligned} \mathbf{if_then_else}\,\mathbf{T}\;a\;b &= \underline{(\lambda pxy.pxy)\mathbf{T}}\;a\;b \\ &\to_\beta \underline{(\lambda xy.\mathbf{T}xy)\;a}\;b \\ &\to_\beta \underline{(\lambda y.\mathbf{T}\;a\;y)\;b} \\ &\to_\beta \mathbf{T}\;a\;b \\ &= \underline{(\lambda xy.x)\;a}\;b \\ &\to_\beta \underline{(\lambda y.a)\;b} \\ &\to_\beta a \end{aligned}$$

Programmieren im λ-Kalkül (2)

Boolesche Funktionen (Forts.):

Konjunktion / Disjunktion / Negation

$$\begin{aligned} \textbf{and} &= \lambda pq.pqp \\ \textbf{or} &= \lambda pqp.ppq \\ \textbf{not} &= \lambda pab.pba \end{aligned}$$

Beispiel:

$$\begin{aligned} \textbf{and}\,\textbf{T}\,\textbf{F} &= \underline{(\lambda pq.pqp)\,\textbf{T}\,\textbf{F}} \\ &\rightarrow_\beta \underline{(\lambda q.\textbf{T}\,q\,\textbf{T})\,\textbf{F}} \\ &\rightarrow_\beta \textbf{T}\,\textbf{F}\,\textbf{T} \\ &= \underline{(\lambda xy.x)\,\textbf{F}\,\textbf{T}} \\ &\rightarrow_\beta \underline{(\lambda y.\textbf{F})\,\textbf{T}} \\ &\rightarrow_\beta \textbf{F} \end{aligned}$$

Programmieren im λ-Kalkül (3)

Boolesche Funktionen (Forts.):

Zur Erinnerung: $\textbf{not} = \lambda pab.pba$, $\textbf{T} = \lambda xy.x$, $\textbf{F} = \lambda xy.y$

Beispiel:

$$
\begin{array}{rcl}
\textbf{not}\ \textbf{T} & = & (\lambda pab.pba)\ \textbf{T} \\
 & \to_\beta & \lambda ab.\textbf{T}ba \\
 & = & \lambda ab.\underline{(\lambda xy.x)ba} \\
 & \to_\beta & \lambda ab.\underline{(\lambda y.b)a} \\
 & \to_\beta & \lambda ab.b \\
 & \to_\alpha & \lambda xy.y \quad \text{(Variablen-Umbenennung)} \\
 & = & \textbf{F}
\end{array}
$$

Programmieren im λ-Kalkül (4)

Arithmetische Funktionen:

Zahlendarstellung: Church-Numerale

$$
\begin{array}{rcl}
0 & = & \lambda fx.x \\
1 & = & \lambda fx.fx \\
2 & = & \lambda fx.f(fx) \\
3 & = & \lambda fx.f(f(fx)) \\
 & \vdots &
\end{array}
$$

n ist durch n-fache Anwendung $f^{(n)}$ einer bel. Funktion f repräsentiert

Nachfolger / Addition / Multiplikation

$$
\begin{array}{rcl}
\textbf{succ} & = & \lambda nfx.f(nfx) \\
\textbf{add} & = & \lambda mnfx.mf(nfx) \\
\textbf{mult} & = & \lambda mnf.m(nf)
\end{array}
$$

add beruht auf der Beobachtung $f^{(m)} \circ f^{(n)} = f^{(m+n)}$

Programmieren im λ-Kalkül (5)

Beispiel:

$$\begin{aligned}
\textbf{succ}\,2 &= \underline{(\lambda nfx.f(nfx))\,2} \\
&\to_\beta \lambda fx.f(2fx) \\
&=_\alpha \lambda fx.f(\underline{(\lambda f'x'.f'(f'x'))fx}) \\
&\to_\beta \lambda fx.f(\underline{(\lambda x'.f(fx'))x}) \\
&\to_\beta \lambda fx.f(f(fx)) = 3
\end{aligned}$$

Beispiel:

$$\begin{aligned}
\textbf{add}\,3\,2 &= \underline{(\lambda mnfx.mf(nfx))\,3}\,2 \\
&\to_\beta \underline{(\lambda nfx.3f(nfx))\,2} \\
&\to_\beta \underline{\lambda fx.3f(2fx)} \\
&=_\alpha \lambda fx.3f(\underline{(\lambda f'x'.f'(f'x'))fx}) \\
&\to_\beta \lambda fx.3f(\underline{\lambda x'.f(fx'))x} \\
&\to_\beta \lambda fx.3f(f(fx)) \\
&=_\alpha \lambda fx.\underline{(\lambda f'x'.f'(f'(f'x')))f}\,(f(fx)) \\
&\to_\beta \lambda fx.\underline{(\lambda x'.f(f(fx')))(f(fx))} \\
&\to_\beta \lambda fx.f(f(f(f(fx)))) = 5
\end{aligned}$$

Programmieren im λ-Kalkül (6)

Beispiel:

$$\begin{array}{rcl}
\textbf{mult}\,2\,3 & = & \underline{(\lambda mnf.m(nf))\,2}\,3 \\
 & \to_\beta & \underline{(\lambda nf.2(nf))\,3} \\
 & \to_\beta & \underline{\lambda f.2(3f)} \\
 & =_\alpha & \lambda f.2(\underline{\lambda f'y.f'(f'(f'y))f}) \\
 & \to_\beta & \lambda f.2(\lambda y.f(f(fy))) \\
 & =_\alpha & \lambda f.\underline{(\lambda f'z.f'(f'z))(\lambda y.f(f(fy)))} \\
 & \to_\beta & \lambda f.(\lambda z.(\lambda y.f(f(f(y)))(\underline{\lambda y.f(f(fy))z}))) \\
 & \to_\beta & \lambda f.(\lambda z.\underline{(\lambda y.f(f(f(y)))(f(f(fz)))})) \\
 & \to_\beta & \lambda f.(\lambda z.f(f(f(f(f(fz)))))) \\
 & = & \lambda fz.f(f(f(f(f(fz))))) \\
 & =_\alpha & \lambda fx.f(f(f(f(f(fx))))) = 6
\end{array}$$

Y-Kombinator: Das coolste Programmierkonstrukt der Welt?!

Siehe `https://www.youtube.com/watch?v=BC8ZAMwfwi4`:

Rekursion via Fixpunkt-Kombinator

β-Äquivalenz

Zwei λ-Terme M, M' heißen β-äquivalent ($M =_\beta M'$) wenn M durch $n \geq 0$, eventuell inverse, β-Reduktionen zu M' transformiert werden kann.

Fixpunkt

N heißt Fixpunkt von F wenn $FN =_\beta N$.

Theorem (Fixpunkt-Satz)

Jeder λ-Term hat einen Fixpunkt.

Beweis:

Der Kombinator $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ ist ein Fixpunkt-Generator:
Es gilt $\mathbf{Y}F =_\beta F\mathbf{Y}F$ für jeden λ-Term F.

Bedeutung: **Y** erlaubt die Darstellung beliebiger Rekursionen!
Für mehr Details siehe, z.B., `https://arxiv.org/abs/0804.3434`

Äquivalenz von Turingmaschinen und λ-Kalkül

Church-Numerale erlauben es arithmetische Funktionen, d.h. Funktionen vom Typ $\mathbb{N}^k \to \mathbb{N}$ als λ-Terme zu codieren.
Church schuf sein Berechnungsmodell einige Monate vor Turing (1936).

Church-Turing-These (Variante 4 – Church)

Wenn eine arithmetische Funktion informell berechenbar ist, dann lässt sie sich als λ-Term darstellen und im λ-Kalkül berechnen (= λ-definierbar).

Ein wichtiger Schritt zur Erhärtung der These:

Theorem (Turing, 1936)

Eine arithmetische Funktion ist genau dann λ-definierbar, wenn sie Turing-berechenbar ist.

Damit ist von zumindest zwei sehr unterschiedlichen Programmier-Paradigmen gezeigt, dass sie prinzipiell gleich ausdrucksstark sind.

Teil 2b – Berechenbarkeit (Überblick)

Teil 2b.1: Die Church-Turing These

Teil 2b.2: Imperatives Paradigma: Turingmaschinen und mehr

Teil 2b.3: Funktionales Paradigma: λ-Kalkül

Teil 2b.4: Rekursive Funktionen

Teil 2b.5: Der Satz von Rice und seine Anwendung

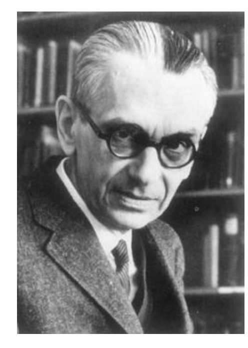
Teil 2b.6: Weitere unentscheidbare Probleme

Teil 2b.7: Unkonventionelle Berechenbarkeitsmodelle

Kurt Gödel – Ein Logiker mit großer Folgewirkung

"Either mathematics is too big for the human mind or the human mind is more than a machine."

- Kurt Gödel (1906 - 1978)
- österreichisch-amerikanischer Logiker
- berühmt (u.a.) für den Beweis der Vollständigkeit der Prädikatenlogik (1929) und seine Unvollständigkeitssätze (1931), die das Hilbertsche Programm unterminieren
- wurde vom TIMES Magazine zu einer der 100 bedeutendsten Personen des 20.Jh. erklärt



Kurt Gödel: über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. in: Monatshefte für Mathematik und Physik 38, 1931, 173 – 198.

Rózsa Péter – Pionierin der Theorie rekursiver Funktionen

"When I began my college education, I still had many doubts about whether I was good enough for mathematics. Then a colleague said the decisive words to me: it is not that I am worthy to occupy myself with mathematics, but rather that mathematics is worthy for one to occupy oneself with."

- Rósza Péter (1905 - 1977)
- ungarische Mathematikerin und Logikerin
- Begründerin der Theorie rekursiver Funktionen
- inspiriert von Gödels Unvollständigkeitssätzen entwickelte sie eine eigenen Zugang zu Berechenbarkeit und Unentscheidbarkeit

Rózsa Péter: Rekursive Funktionen, Budapest 1951. (Monographie, mehrere Übersetzungen und weitere Auflagen)

Ein klassischer Zugang zu Berechenbarkeit

Grundidee (induktive Charakterisierung von Berechenbarkeit):
Man fängt mit einigen offensichtlich berechenbaren Grundfunktionen an und definiert möglichst wenige, schlanke Operatoren (Funktionalen), die gegebene Funktionen in ebenfalls berechenbare Funktionen abbilden.

Wir betrachten (o.B.d.A.!) Funktionen $\mathbb{N}^k \to \mathbb{N}$, $k \geq 0$

Grundfunktionen

- konstante Funktionen C_n^k: $C_n^k(x_1, ..., x_k) = n$
- Nachfolger-Funktion S: $S(x) = x + 1$
- Projektions-Funktionen P_i^k: $P_i^k(x_1, ..., x_k) = x_i$ $(1 \leq i \leq k)$

Beispiele:
C_2^1 $[f(x) = 2, \lambda x.2]$ C_0^3 $[f(x, y, z) = 0, \lambda xyz.0]$ C_2^0 $[2]$ (C_3^0, C_4^0) $[(3, 4)]$
S $[f(x) = x + 1, \mathbf{succ}]$ (S, C_2^1) $[f(x) = (x + 1, 2)]$
P_1^3 $[f(x, y, z) = x, \lambda xyz.x]$ P_3^3 $[f(x, y, z) = z, \lambda xyz.z]$ P_1^1 $[f(x), \lambda x.x]$
(P_2^2, P_1^2) $[f(x, y) = (y, x)]$ (P_3^3, P_2^3, P_2^3) $[f(x, y, z) = (z, y, y)]$

Einfache Zusammensetzung von Funktionen

Komposition $\circ$ (auch: Einsetzungs-Operator)

Gegeben $h : \mathbb{N}^m \to \mathbb{N}$ und für $1 \leq i \leq m$: $g_i : \mathbb{N}^k \to \mathbb{N}$, ist

$$f = (h \circ (g_1, ..., g_m)) : \mathbb{N}^k \to \mathbb{N}$$

definiert durch $f(x_1, ..., x_k) = h(g_1(x_1, ..., x_k), \ldots g_m(x_1, ..., x_k))$.

Links-Assoziativität: $A \circ B \circ C = (A \circ B) \circ C$ (also wie bei λ-Termen)

Beispiele:

$S \circ P_2^2$ $[f(x, y) = y + 1]$ $\quad$ $C_0^1 \circ S$ $[f(x) = 0]$ $\quad$ $S \circ C_0^1$ $[f(x) = 1]$

$S \circ S \circ S$ $[f(n) = n + 3]$ $\quad$ $S \circ S \circ C_0^0$ $[2]$

$P_2^2 \circ (C_1^1, S)$ $[f(x) = x + 1]$

$$\begin{aligned} P_2^2 \circ (S \circ S, S) \circ (S \circ C_0^1) \circ P_1^1 &= (P_2^2 \circ (S \circ S, S)) \circ (S \circ C_0^1) \\ &= S \circ (S \circ C_0^1) \\ &= S \circ C_1^1 = C_2^1 \end{aligned}$$

Beachte: Die Sprache der Funktionsausdrücke ist variablenfrei!

Einfache (‘primitive’) Rekursion

Beobachtung: Eine induktive (= rekursive) Definition einer Funktion bezieht sich auf 2 Bestandteile: Anfangsfunktion und Übergangsfunktion.

Beispiel: rekursive Definition von Addition

Anfang: $Add(0, n) = n$ $[0 + n = n]$

Übergang: $Add(S(m), n) = S(Add(m, n))$ $[(m + 1) + n = (m + n) + 1]$

Übergangsfunktion kann sich auch auf das Argument n beziehen:

Beispiel: rekursive Definition von Multiplikation

Anfang: $Mult(0, n) = 0$ $[0 \cdot n = 0]$

Übergang: $Mult(S(m), n) = Add(n, Mult(m, n))$ $[(m + 1) \cdot n = m \cdot n + n]$

Übergangsfunktion kann sich auch auf die Rekursionstiefe m beziehen:

Beispiel: Die Faktorielle (auch: Fakultäts-Funktions)

Anfang: $Fact(0) = 1$ $[0! = 1]$

Übergang: $Fact(S(m)) = Mult(S(m), Fact(m))$ $[(m + 1)! = (m + 1) \cdot m!]$

Der Operator **Pr** der primitiven Rekursion

Primitive Rekursion **Pr**

Gegeben $g : \mathbb{N}^k \to \mathbb{N}$ und $h : \mathbb{N}^{k+2} \to \mathbb{N}$, ist

$$f = \mathbf{Pr}(g, h) : \mathbb{N}^{k+1} \to \mathbb{N}$$

definiert durch $f(0, x_1, ..., x_k) = g(x_1, ..., x_k)$ und
$f(m+1, x_1, ..., x_k) = h(m, f(m, x_1, ..., x_k), x_1, ..., x_k)$.

Die mit **Pr** und $\circ$ aus Grundfunktionen definierbaren Funktionen heißen primitiv-rekursive Funktionen.

Beispiel: $Add = \mathbf{Pr}(P_1^1, S \circ P_2^3)$

$$\begin{aligned}
Add(0,5) &= P_1^1(5) = 5\\
Add(1,5) &= S \circ P_2^3(0,5,5) = S(5) = 6\\
Add(2,5) &= S \circ P_2^3(1,6,5) = S(6) = 7\\
Add(3,5) &= S \circ P_2^3(2,7,5) = S(7) = 8
\end{aligned}$$

Primitive Rekursion – weitere Beispiele

Multiplikation:

$Mult = \mathbf{Pr}(C^1_0, Add \circ (P^3_2, P^3_3)) = \mathbf{Pr}(C^1_0, \mathbf{Pr}(P^1_1, S \circ P^3_2) \circ (P^3_2, P^3_3))$

$Mult(m, n)$: C^1_0 bezieht sich auf n, $Add \circ (P^3_2, P^3_3)$ auf $(m, Mult(m, n), n)$

Faktorielle:

$Fact = \mathbf{Pr}(C^0_1, Mult \circ (P^2_1, S \circ P^2_2)) = \mathbf{Pr}(C^0_1, \mathbf{Pr}(C^1_0, \mathbf{Pr}(P^1_1, S \circ P^3_2 ...$

$Fact(m)$: C^0_1 hat kein Argument (daher ist der obere Index 0)

$Mult \circ (P^2_2, S \circ P^2_1)$ bezieht sich auf $(m, Fact(m))$:

$Mult \circ (P^2_2, S \circ P^2_1)(m, Fact(m)) = Mult \circ (Fact(m), m+1) = Fact(m) \cdot (m+1)$

Abgeschnittene Vorgängerfunktion:

$APred = \mathbf{Pr}(C^0_0, P^2_1)$, weil $APred(0) = 0$ und $APred(m + 1) = m$

$APred(5) = \mathbf{Pr}(C^0_0, P^2_1)(4 + 1) = P^2_1(4, APred(4)) = 4$

Charakteristische Funktion für '$\neq 0$' (Signum):

$Signum = \mathbf{Pr}(C^0_0, C^2_1)$, weil $Signum(0) = 0$ und $Signum(m + 1) = 1$

Abgeschnittene Subtraktion:

$ASub = \mathbf{Pr}(P^1_1, APred \circ P^3_2) \circ (P^2_2, P^2_1)$ $\quad ASub(m, n) = \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Boolesche Funktionen / Fallunterscheidung

Wir verwenden die übliche Codierung: 1 für **wahr** / 0 für **falsch**

Negation / Konjunktion / Disjunktion:
$Not = Asub \circ (C_1^1, P_1^1)$ $[Not(x) = Asub(1, x)]$
$And = Mult$ $[And(x, y) = 1 \iff x \cdot y = 1$, 0 sonst]
$Or = Signum \circ Add$ $[Or(x, y) = 1 \iff x + y = 0$, 1 sonst]

Charakterische Funktion $\chi(<)$ für die ‘kleiner’-Relation:
$less = Signum \circ ASub \circ (P_2^2, P_1^2)$ $[less(m, n) = 1 \iff m < n$, 0 sonst]

Beachte: Alle Booleschen Ausdrücke über Ungleichungen sind ausdrückbar!

Definition durch Fallunterscheidung:

$$f(m, n) = \begin{cases} g(m, n) & \text{falls } m < n \\ h(m, n) & \text{sonst} \end{cases}$$

G bzw. H seien Funktionsausdrücke für g bzw. h. Dann ist

$$Add \circ (Mult \circ (G, less),\ Mult \circ (H, Not \circ less))$$

ein Funktionsausdruck für $f = g(m, n) \cdot \chi(m < n) + h(m, n) \cdot \chi(m \geq n)$

Minimierung – μ- und $\overline{\mu}$-Operator

Beobachtung: Primitiv rekursive sind total (überall definiert).
Für, z.B., echte Subtraktion (in $\mathbb{N}$) benötigen wir einen weiteren Operator.

μ-und $\overline{\mu}$-Rekursion

Gegeben eine totale Funktion $g : \mathbb{N}^{k+1} \to \mathbb{N}$, sind

$$f = \mu g : \mathbb{N}^k \to \mathbb{N} \quad \text{und} \quad f' = \overline{\mu} g : \mathbb{N}^{k+1} \to \mathbb{N}$$

definiert durch $f(x_1, ..., x_k) = \min_{y \geq 0}[g(y, x_1, ..., x_k) = 0]$ bzw.
durch $f'(x_1, x_2, ..., x_{k+1}) = \min_{y \geq 0}[g(y, x_2, ..., x_{k+1}) = x_1]$.

$(\mu h)(x_1, ..., x_k) = (\overline{\mu} h)(0, x_1, x_2, ..., x_k)$
Umgekehrt lässt sich $\overline{\mu} f$ mit Hilfe von μ, **Pr** und $\circ$ darstellen (aufwändig).

Die mit μ (äquivalent: $\overline{\mu}$), **Pr** und $\circ$ aus Grundfunktionen definierbaren Funktionen heißen μ-rekursive oder auch partiell rekursive Funktionen.
(Das entsprechende Minimum muss nicht existieren, daher 'partiell'.)
Wenn eine solche Funktion überall definiert ist, heißt sie total rekursiv.

Minimierung – Beispiele

Beispiele

$(\mu P^1_1)(n) = \min_{y\geq 0}[y{=}0] = 0$, daher $\mu P^1_1 = C^0_0$

$(\overline{\mu} P^1_1)(n) = \min_{y\geq 0}[y{=}n] = n$, daher $\overline{\mu} P^1_1 = P^1_1$

$(\mu S)(n) = \min_{y\geq 0}[y + 1 = 0]$ ist immer undefiniert, ebenso μC^1_k für $k \geq 1$

$(\overline{\mu} S)(n) = \min_{y\geq 0}[y + 1 = n]$ ist die (echte) Vorgänger-Funktion,
also undefiniert für $n = 0$

$(\overline{\mu} Add)(m, n) = \min_{y\geq 0}[y + n = m] = m - n$, undefiniert für $m < n$
also ist $Sub = \overline{\mu} Add$ die (echte) Subtraktion in $\mathbb{N}$

$(\overline{\mu} Mult)(m, n) = \min_{y\geq 0}[y \cdot n = m] = \frac{m}{n}$, falls m durch n teilbar ist
und undefiniert sonst
also ist $Div = \overline{\mu} Mult$ die (echte) Division in $\mathbb{N}$

Partielle Rekursivität und Berechenbarkeit

Beobachtung:
Jede primitiv rekursive Funktion ist berechenbar; z.B., mit SIMPLE.
Auch jede partiell rekursive Funktion ist berechenbar, da μ und $\overline{\mu}$ nur auf totale (berechenbare) Funktionen angewendet werden. Die Suche nach der minimalen Lösung terminiert aber nicht immer.

Umgekehrt gilt auch:

Theorem (Turing, Kleene, Peter)

Jede auf einer TM berechenbare Funktion ist partiell rekursiv.

Daraus ergibt sich:

Church-Turing-These (Variante 5 – Gödel/Kleene/Church/Peter)

Die Klasse der informell berechenbaren Funktionen stimmt mit der Klasse der partiell rekursiven Funktionen überein.

Berechenbar und total, aber nicht primitiv rekursiv

Beobachtungen:

- Die Funktionsausdrücke, die durch Anwendung von **Pr** und $\circ$ auf Grundfunktionen entstehen lassen sich systematisch aufzählen.
  $F_0, F_1, F_2, \ldots$ seien die entsprechenden Funktionen vom Typ $\mathbb{N} \to \mathbb{N}$.
- Alle Einträge in folgender (unendlichen) Tabelle sind berechenbar:

	F_0	F_1	F_2	F_3	$\ldots$
0	$F_0(0)$	$F_1(0)$	$F_2(0)$	$F_3(0)$	$\ldots$
1	$F_0(1)$	$F_1(1)$	$F_2(1)$	$F_3(1)$	$\ldots$
2	$F_0(2)$	$F_1(2)$	$F_2(2)$	$F_3(2)$	$\ldots$
3	$F_0(3)$	$F_1(3)$	$F_2(3)$	$F_3(3)$	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

- Cantor'sches Diagonalverfahren: Die Funktion $D(n) = F_n(n) + 1$ ist berechenbar, kann aber für kein i mit F_i übereinstimmen.

Konsequenzen für die Informatik

Aus unseren Beobachtungen ergibt sich:

Theorem

Nicht alle totalen berechenbaren Funktionen sind primitiv rekursiv sind.

Allgemeiner folgt (via Church-Turing-These):

Garantierte Termination erzwingt Unvollständigkeit

Es kann keine Programmiersprache geben, in der alle Programm terminieren und in der alle totale und berechenbaren Funktionen ausdrückbar sind.

Anmerkungen:

- Man benützt hier, dass sich Programme immer systematisch aufzählen lassen müssen. Sonst könnte man für diese Sprache auch keinen Parser (Compiler / Interpreter) schreiben.
- Es gibt Programmiersprachen (bzw. Umgebungen), in denen nur terminierende Programme formuliert werden können. Darin ist aber eben nicht jeder terminierende Algorithmus ausdrückbar.

Eine total rekursive, aber nicht primitiv rekursive Funktion

Ackermann-Funktion

Die Funktion $Ack : \mathbb{N}^2 \to \mathbb{N}$ ist durch folgende Gleichungen definiert:

$$\begin{aligned} Ack(0, n) &= n + 1 \\ Ack(m + 1, 0) &= Ack(m, 1) \\ Ack(m + 1, n + 1) &= Ack(m, Ack(m + 1, n)) \end{aligned}$$

Es ist nicht ganz offensichtlich, dass *Ack* berechenbar ist!
Übungsaufgabe: Berechnen Sie $Ack(3, 0)$!

Es gilt:

$$\begin{aligned} Ack(1, n) &= n + 2 \\ Ack(2, n) &= 2n + 3 \\ Ack(3, n) &= 2^{n+3} - 3 \\ Ack(4, n) &= \underbrace{2^{2^{\cdots^{2}}}}_{n+3 \text{ Zweier}} - 3 \end{aligned}$$

Theorem

Ack ist berechenbar und total, aber nicht primitiv rekursiv.

Teil 2b – Berechenbarkeit (Überblick)

Teil 2b.1: Die Church-Turing These

Teil 2b.2: Imperatives Paradigma: Turingmaschinen und mehr

Teil 2b.3: Funktionales Paradigma: λ-Kalkül

Teil 2b.4: Rekursive Funktionen

Teil 2b.5: Der Satz von Rice und seine Anwendung

Teil 2b.6: Weitere unentscheidbare Probleme

Teil 2b.7: Unkonventionelle Berechenbarkeitsmodelle

Noch einmal: Entscheidungsprobleme

Zur Erinnerung (Teil 1 und 2a):
Wir identifizieren (Entscheidungs)probleme mit Sprachen ($\subseteq \Sigma^*$)

Beispiele: Menge der haltenden Programme, gültigen PL-Formeln, etc.
Ist ein beliebige(s) Programm, PL-Formel, ... s in der jeweiligen Menge X?

(Un)entscheidbar heißt:
Es gibt (k)ein haltendes Programm, das $s \in X$? immer korrekt beantwortet.

Wir haben bisher einzelne Probleme X *case-by-case* als unentscheidbar erkannt. (Wenn X und $\overline{X}$ beide unentscheibar sind, dann ist zumindest eines der Probleme X bzw. $\overline{X}$ nicht einmal semi-entscheidbar.)
Beweismethode: Reduktion eines bereits als unentscheidbar erkannten Problems (typischerweise des Halteproblems) auf X.

Herausforderung:
Finde eine Methode, die es erlaubt ganze Klassen von Problemen als unentscheidbar zu erkennen ohne neue Reduktionen erfinden zu müssen.

Eigenschaften von Sprachen

Wie bereits mehrfach betont, müssen 2 Ebenen strikt getrennt bleiben:

(1): Sprachen (= Problemen), Funktionen, (Input/Output-)Relationen,...
(2): Grammatiken, Programme, Turingmaschinen (TMs), λ-Terme,...

Ebene 2 bilden Darstellungsmittel, Ebene 1 die dargestellten Objekte.
Entscheidungsprobleme beziehen sich auf Ebene 1 und fragen, ob es entsprechende Entscheidungsmechanismen der Ebene 2 gibt.

Wir konzentrieren uns zunächst in der Ebene 1 auf Sprachen $\subseteq \Sigma^*$.

Um die **Herausforderung** annehmen zu können benötigen wir:

(Extensionale) Eigenschaften rekursiv aufzählbarer Sprache

Eine (Sprach-)Eigenschaft ist eine Teilmenge $\mathcal{E}$ der rekursiv aufzählbaren Sprachen über einem geeigneten Alphabet Σ. (Also $\mathcal{E} \subsetneq \mathcal{P}(\Sigma^*)$.)
$\mathcal{E}$ heißt trivial, wenn sie entweder leer ist oder aus allen rekursiv aufzählbaren Sprachen (über Σ) besteht.

Beispiele von Sprach-Eigenschaften

Beispiele für Eigenschaften rekursiv aufzählbarer Sprachen

$\mathcal{E}_1 = \{L \mid L \text{ ist entscheidbar } \}$
$\mathcal{E}_2 = \{L \mid L \text{ ist semi-entscheidbar } \}$
$\mathcal{E}_3 = \{L \mid L \text{ ist endlich } \}$
$\mathcal{E}_4 = \{L \mid L = L^*\}$
$\mathcal{E}_5 = \{\{\varepsilon\}\}$
$\mathcal{E}_6 = \{\}$
$\mathcal{E}_7 = \{\{\}\}$
$\mathcal{E}_8 = \{L \mid L = \{a, b\}^* \text{ oder } L = \{a\}^*\}$

Nur $\mathcal{E}_2$ und $\mathcal{E}_6$ sind triviale Eigenschaften.

Beispiele für Klassifikationen

$\{a\}^*$ hat die Eigenschaften $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_4, \mathcal{E}_8$, aber nicht $\mathcal{E}_3, \mathcal{E}_5, \mathcal{E}_6, \mathcal{E}_7$.
$\{\varepsilon, a, aa\}$ hat die Eigenschaften $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$, aber nicht $\mathcal{E}_4, \mathcal{E}_5, \mathcal{E}_6, \mathcal{E}_7, \mathcal{E}_8$.
$\{\langle M\rangle \mid \langle M\rangle \text{ ist der Code einer TM } M\}$ hat nur die Eigenschaften $\mathcal{E}_1, \mathcal{E}_2$.

Der Satz von Rice[‡] – Ein mächtiges Werkzeug

Theorem (Satz von Rice – informelle Kurzform)

Jede nicht-triviale Sprach-Eigenschaft ist unentscheidbar.

Beachte: Genau genommen können nur Sprachen (un)entscheidbar sein. Eigenschaften sind aber Mengen von Sprachen. Diese können indirekt, über ihre jeweilige Darstellung als TM, als Sprachen repräsentiert werden.

Definition (Sprach-Code einer Eigenschaft)

Für jede Eigenschaft $\mathcal{E}$ rekursiv aufzählbarer Sprachen sei $L_{\mathcal{E}} = \{\langle M \rangle \mid L(M) \in \mathcal{E}\}$, also die Menge aller Codes von jenen TMs, die eine Sprache $L \in \mathcal{E}$ akzeptieren.

(Un-)Entscheidbarkeit von $\mathcal{E}$ meint (Un-)Entscheidbarkeit von $L_{\mathcal{E}}$.

Theorem (Satz von Rice – genauere Formulierung)

Für jede nicht-triviale Eigenschaft $\mathcal{E}$ rekursiv aufzählbarer Sprachen ist $L_{\mathcal{E}}$ unentscheidbar. (Kurz: '$\mathcal{E}$ ist unentscheidbar'.)

[‡] Henry Gordon Rice (1920-2003), Dissertation 1951, 1953 publiziert

Der Satz von Rice: Anwendungen

Einige Beispiele von Eigenschaften, für die der Satz von Rice unmittelbar die Unentscheidbarkeit impliziert.

- Es ist nicht entscheidbar, ob eine Turingmaschine mehr als fünf Wörter akzeptiert. (Sprach-Eigenschaft $\mathcal{E} = \{L \mid |L| > 5\}$).
  Um den Satz von Rice anwenden zu können, müssen wir noch überprüfen, ob die Eigenschaft $\mathcal{E}$ nicht-trivial ist. Wir müssen also mindestens eine rekursiv aufzählbare Sprache finden, die die Eigenschaft erfüllt, und eine, die die Eigenschaft nicht erfüllt: z.B. $L_1 = \{a, a^2, a^3, \dots, a^6\} \in \mathcal{E}$ und $L_2 = \{\} \notin \mathcal{E}$.
- Es ist nicht entscheidbar, ob eine Turingmaschine nur endlich viele Wörter akzeptiert. (Sprach-Eigenschaft $\mathcal{E} = \{L \mid L \text{ ist endlich }\}$) Kurz: Es ist nicht entscheidbar, ob eine Sprache endlich ist.
- Es ist nicht entscheidbar, ob eine (von einer Turingmaschine akzeptierte) Sprache regulär (kontextfrei, kontextsensitiv) ist.
- Es ist nicht entscheidbar, ob eine Turingmaschine überhaupt etwas akzeptiert. (Auch die Spracheigenschaft $\mathcal{E} = \{\{\}\}$ ist nicht-trivial!)

Nicht-Anwendbarkeit des Satzes von Rice – Intensional versus Extensional

Es gibt durchaus entscheidbare Eigenschaften für Turingmaschinen:
Z.B. ist es entscheidbar, ob eine TM mindestens 5 Zustände hat.
Das ist keine Eigenschaft der akzeptierten Sprache, sondern der TM selbst.

Intensionale Eigenschaften:
beziehen sich auf die Darstellungsmittel, hier TMs ($\rightarrow$ Folie 62, Ebene 2)

Extensionale Eigenschaften:
beziehen sich auf das Dargestelle, hier Sprachen ($\rightarrow$ Folie 62, Ebene 1)

Beachte: Sprach-Eigenschaften sind extensional. Der Satz von Rice bezieht sich also nur auf extensionale, nicht auf intensionale Eigenschaften!

Auch intensionale Eigenschaften können unentscheidbar sein.
Z.B.: Akzeptiert eine vorgelegte TM ihren eigenen Code?

Der Satz von Rice für Funktionen

Nicht nur Eigenschaften von Sprachen, sondern auch Eigenschaften berechenbarer Funktionen sind relevant.

(Extensionale) Eigenschaften berechenbarer Funktionen

Eine (extensionale) Funktions-Eigenschaft ist eine Teilmenge $\mathcal{F}$ der Turing-berechenbaren Funktionen (eines bestimmten Typs).
$\mathcal{F}$ heißt trivial, wenn sie entweder leer ist oder aus allen Turing-berechenbaren Funktionen (des entsprechenden Typs) besteht.

Definition (Sprach-Code einer Funktions-Eigenschaft)

Für jede Funktions-Eigenschaft $\mathcal{F}$ sei $L_{\mathcal{F}} = \{\langle M \rangle \mid F(M) \in \mathcal{F}\}$, also die Menge aller Codes von TMs, die eine Funktion $f \in \mathcal{F}$ berechnen.

Theorem (Satz von Rice für Funktionen)

Für jede nicht-triviale Eigenschaft $\mathcal{F}$ von Turing-berechenbaren Funktionen ist die Sprache $\mathcal{C}(\mathcal{F})$ unentscheidbar. (Kurz: '$\mathcal{F}$ ist unentscheidbar'.)

Anwendungen des Satzes von Rice für Funktionen

Beispiele:

- Es ist nicht entscheidbar, ob eine Turingmaschine M die Nachfolger-Funktion berechnet.
  (Die Funktions-Eigenschaft $\{f : \mathbb{N} \to \mathbb{N} \mid f(x) = x + 1\}$ ist nicht-trivial.)
- Es ist nicht entscheidbar, ob eine Turingmaschine nur totale arithmetische Funktionen berechnet. (Totalität ist eine nicht-triviale Eigenschaft, weil es totale Funktion gibt die Turing-berechenbar sind, aber nicht alle Turing-berechenbare Funktionen total sind.)
- Es ist nicht entscheidbar, ob für die Funktion $F(M) : \{0,1\}^* \to \{0,1\}^*$ einer Turingmaschine M gilt, dass $F(M)(1) = 111$ oder $F(M)(1) = 000$. (Die Funktions-Eigenschaft $\{f : \{0,1\}^* \to \{0,1\}^* \mid f(1) = 111 \text{ oder } f(1) = 000\}$ ist offensichtlich nicht-trivial.)

Weiteres zur Anwendbarkeit des Satzes von Rice

Beobachtung:
Die Unentscheidbarkeit des Korrektheitsproblems aus Teil 1, angewendet auf arithmetische Funktionen, folgt unmittelbar aus dem Satz von Rice. Das gilt auch für die meisten verwandten Probleme: Z.B. das universelle Halteproblem oder das Äquivalenzproblem.

Anmerkung:
Die Beschränkung auf Turingmaschinen als Darstellungsmittel für Sprachen bzw. Funktionen ist unwesentlich. Analoge Sätze gelten auch für übliche Programmiersprachen, Typ-0-Grammatiken, λ-Terme, etc.

Achtung: Es ist nicht immer einfach festzustellen, ob eine Funktions- oder Sprach-Eigenschaft trivial ist. Z.B. ist es trivial entscheidbar, ob eine die von einer TM berechnete Funktion (vom Typ $\mathbb{N}^k \to \mathbb{N}$) λ-definierbar ist.

Die Antwort ist immer 'ja': Jede Turing-berechenbare Funktion ist λ-definierbar. Aber das beruht auf tiefen, keineswegs trivialen Erkenntnissen von Turing und Kleene!

Teil 2b – Berechenbarkeit (Überblick)

Teil 2b.1: Die Church-Turing These

Teil 2b.2: Imperatives Paradigma: Turingmaschinen und mehr

Teil 2b.3: Funktionales Paradigma: λ-Kalkül

Teil 2b.4: Rekursive Funktionen

Teil 2b.5: Der Satz von Rice und seine Anwendung

Teil 2b.6: Weitere unentscheidbare Probleme

Teil 2b.7: Unkonventionelle Berechenbarkeitsmodelle

Probleme zu formalen Sprachen [teils WH]

. Folgende Problem sind unentscheidbar

- Gegeben: Eine unbeschränkte (Typ-0) Grammatik G und $w \in \Sigma^*$.
  Frage: $w \in L(G)$?
- Gegeben: Eine unbeschränkte Grammatik G.
  Frage: Ist $L(G)$ leer? (Ähnlich: $L(G) = \Sigma^*$)?
- Gegeben: Eine kontextfreie (Typ-2) Grammatik G.
  Frage: Ist $L(G)$ inhärent mehrdeutig? (Siehe Teil 2a, Folie 79)
- Gegeben: Eine kontextfreie Grammatik G.
  Frage: Ist $L(G)$ regulär?
- Gegeben: 2 kontextfreie Grammatiken G_1 und G_2.
  Frage: $L(G_1) = L(G_2)$?
- Gegeben: 2 kontextfreie Grammatiken G_1 und G_2.
  Frage: $L(G_1) \cap L(G_2) = \emptyset$?

Beachte: Der Satz von Rice ist nur auf die ersten 2 Probleme anwendbar!
Überlege: Was folgt [nicht] für reguläre und kontextsensitive Sprachen?

Die Busy-Beaver-Funktion

Definition (Busy-Beaver-Funktion BB)

$BB(n)$ ist die maximale Zahl von 1-en, die eine TM mit n Zuständen und $\Gamma = \{B, 1\}$, angewendet auf das leere Band, bei Termination hinterlässt.

Theorem

BB ist unberechenbar (d.h., nicht Turing-berechenbar).

Anmerkungen:

- BB ist eine totale Funktion vom Typ $\mathbb{N} \to \mathbb{N}$.
- Man kennt die Werte von $BB(n)$ nur für $n = 1, 2, 3, 4$.
- Würde man, z.B., $BB(100)$ kennen, so wären damit viele berühmte offene mathematische Probleme (im Prinzip) gelöst.
- Für $\Gamma = \{B, 0, 1\}$ ist bereits unbekannt was mit nur 3 Zuständen auf einer TM berechenbar ist. Bei 4 Bandsymbolen kann man nicht einmal mehr TMs mit nur 2 Zuständen vollständig analysieren!
- Siehe (z.B.) `https://en.wikipedia.org/wiki/Busy_beaver`

Kolmogorov-Komplexität: Perfekte Komprimierung

Definition (Kolmogorov-Komplexität $K(n)$)

$K(n)$ ist die minimale Länge des Binärcodes einer TM die, angewendet auf das leere Band, $[n]_2$ ausgibt.

Anmerkung: Die exakte Definition bezieht sich auf eine universelle TM.

Theorem

K ist eine (Turing-)unberechenbare Funktion (vom Typ $\mathbb{N} \to \mathbb{N}$).

Anmerkungen:

- Intuition: $K(n)$ gibt an wie knapp man n eindeutig beschreiben kann.
- Das ist auch in der Praxis relevant:
  Stichworte: Optimale Codes, maximaler Informationsgeshalt,...
- Trotz der Unberechenbarkeit lässt sich viel allgemeines über K sagen:
  Z.B.: Es gibt ein c, sodass $K(n) \leq \log_2(n) + c$ für alle n,
  andererseits $K(n) > \log_2(n) - c$ für ‘die allermeisten’ n.

Das Postsche Korrespondenzproblem

- Gegeben: Endliche Menge von Wort-Paaren
  $$\{(v_i, w_i) \mid 1 \leq i \leq n,\ v_i, w_i \in \Sigma^*\}$$
  Frage: Gibt es eine Folge $i_1, \ldots, i_k$, wobei $i_\ell \in \{1, \ldots, n\}$ für $1 \leq \ell \leq k$, sodass $v_{i_1} \ldots v_{i_k} = w_{i_1} \ldots w_{i_k}$?

Theorem

Das Postsche Korrespondenzproblem ist unentscheidbar.

Beispiel:
Gegeben: $\{(1, 101), (10, 00), (011, 11)\}$, wobei
$v_1 = 1,\ v_2 = 10,\ v_3 = 011,\ \ w_1 = 101,\ w_2 = 00,\ w_3 = 11$
Eine Lösung: $1, 3, 2, 3$, da $1 \cdot 011 \cdot 10 \cdot 011 = 101 \cdot 11 \cdot 00 \cdot 11$

Anmerkungen:

- Die kleinste Lösung für $\{(001, 0), (01, 011), (01, 101), (10, 001)\}$ benötigt bereits eine Indexfolge der Länge 66 (also 66 Wort-Paare)!
- Für $\Sigma = \{1\}$ ist das Postsche Korrespondenzproblem noch entscheidbar; aber bereits unentscheidbar für $\Sigma = \{0, 1\}$

Hinweise zu anderen unentscheidbarer Probleme (als Anregung Details dazu selbst zu erforschen!)

- Wortproblem für (endlich erzeugte) Gruppen
- Tiling: Gegeben Quadrate mit verschiedenen Kantenfarben, kann man die gesamte Ebene so belegen, dass alle Kantenfarben passen?
- Conway's Game of Life: Sind bestimmte Konfigurationen erreichbar
- Zellular-Automaten: (Z.B.) Periodizität des Ablaufs?
- Ray-Tracing: Erreicht ein Lichtstrahl, der von einem bestimmten Punkt ausgeht und von den Oberflächen gegebener geometrischer Objekte reflektiert wird ein bestimmtes Areal?
- Optimale Flugrouten unter Berücksichtung realer Kostenregeln
  `http://www.demarcken.org/carl/papers/ITA-software-travel-complexity/ITA-software-travel-complexity.pdf`
- (viele Probleme aus Topologie, Analysis, Physik, Ökonomie, etc.)

Teil 2b – Berechenbarkeit (Überblick)

Teil 2b.1: Die Church-Turing These

Teil 2b.2: Imperatives Paradigma: Turingmaschinen und mehr

Teil 2b.3: Funktionales Paradigma: λ-Kalkül

Teil 2b.4: Rekursive Funktionen

Teil 2b.5: Der Satz von Rice und seine Anwendung

Teil 2b.6: Weitere unentscheidbare Probleme

Teil 2b.7: Unkonventionelle Berechenbarkeitsmodelle

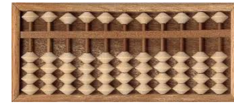
Mechanische Rechner

Mechanische Rechenhilfen und Rechenautomaten sind ein ein zentrales Thema der Geschichte, aber auch der Didaktik der Informatik.

Aus heutiger Sicht: Spezialanwendungen, keine Berechnungs-Universalität

Einige Beispiele:

- Abakus (Rechenbrett):



- Charles Babbage' Difference Engine (1820)†, Analytic Engine (1838)†: → Lady Ada Lovelace wird erste Programmerin
- MONIAC† (Monetary National Income Analogue Computer): auch 'Financephalograph', erfunden 1949 vom neuseeländischen Ökonomen Bill Phillips um die Wirtschaft Großbritanniens zu simulieren.
- Гидравлическиы интегратор† ('Wasser-Integrierer'): Vladimir S. Lukyanov (1936), zur Lösung inhomogener Differentialgleichungen

†click-barer Link

Andere physikalische und biochemische Grundlagen

Stichworte zu den beiden prominentesten Beispielen:

- Qantum Computing†
  - beruht auf Superposition und Verschränkung
  - inhärent probabilistisch: Berechnung relativ zu Fehlergrenzen
  - Formales Modell: QTM, (Quantum-Turing-Maschine)
    modulo beliebig kleiner ϵ-Schranken) kann auf einer QTM dasselbe berechnet und entschieden werden wie auf einer TM
  - in der Praxis: große Herausforderungen durch Dekohärenz
  - Komplexitätsgewinne wohl nur für Spezialanwendungen zu erwarten
- DNA Computing†
  - nützt die komplexen biochemischen Prozesse der Genetik
  - (zumindest in der Theorie:) universell im Sinn der Church-Turing-These
  - massive Parallelität auf kleinstem Raum mit wenig Energie
  - Spezialfall des größeren Gebiets des ‘Molekularen Rechnens’

† click-barer Link

Interaktives Rechnen

Viele aktuelle Anwendungs-Scenarios beruhen auf massiver Interaktion zwischen Rechner und Nutzer – allgemeiner: Umwelt.

Das stellt eine Herausforderung für die Berechenbarkeitstheorie ohne die Church-Turing-These grundsätzlich zu verneinen.

Einige Aspekte aus aktueller Forschung und Praxis:

- Modelle der Kommunikation und Interaktion zwischen Agenten
- Rechner-Umwelt-Interaktion als (formales) Spiel ('Game Semantics')
- Verifikation offener Systeme
- massive, fehlerbehaftete, unsichere Web-Interaktionen

Leseempfehlung:
Dina Goldin, Scott A. Smolka, Peter Wegner (Herausgeber)
Interactive Computation – The New Paradigm, Springer 2006.

6.0 VU Theoretische Informatik (192.017)

Teil 3: Einführung in Komplexitätstheorie

Stefan Woltran

Institut für Logic and Computation

Wintersemester 2023

Teil 3: Einführung in Komplexitätstheorie

Teil 3.1: Grundlegende Konzepte und Wiederholung

Teil 3.2: Die Komplexitätsklassen P und NP

Teil 3.3: NP-Vollständigkeit

Teil 3.4: Weitere Komplexitätsklassen

Wiederholung: Entscheidungsproblem

Definition von “Problemen”

Ein Problem ist definiert durch eine (abzählbar) unendliche Menge von möglichen Instanzen (d.h.: möglichen Inputs) zusammen mit einer Frage.

Ein Entscheidungsproblem ist ein Problem, bei dem die Frage eine ja/nein Antwort erwartet.

Beispiel für ein Problem

GRAPH-ERREICHBARKEIT:
INSTANZ: Ein Graph $G = (V, E)$ und Knoten $u, v \in V$.
FRAGE: Gibt es im Graph G einen Pfad von u nach v?

GRAPH-ERREICHBARKEIT ist ein Entscheidungsproblem.

Wiederholung: Algorithmen

Definition eines Algorithmus

Ein Algorithmus für ein Problem $\mathcal{P}$ ist eine Beschreibung von Rechenschritten, die es uns erlauben, jede beliebige Instanz des Problems $\mathcal{P}$ zu lösen.

- Ein Algorithmus muss auf alle Instanzen des Problems anwendbar sein und dabei ...
  1. nach endlich vielen Schritten terminieren und
  2. die korrekte Antwort auf die Frage liefern.

Wiederholung: Unsere Programmiersprache SIMPLE

- Wir verwenden den Kern von prozeduralen Programmiersprachen:
  - Variablen von unterschiedlichem Typ (String, Integer, Boolean, Void)
  - Wertzuweisungen, einfache Rechenoperationen (z.B.: $x := y + z$)
  - "if/then/else" Anweisungen
  - "while" Schleifen
  - "for" und "repeat" Schleifen
    (können durch "while" Schleifen ersetzt werden)
  - "return" Anweisungen
  - Prozeduren, Funktionen
- Der Input für ein Programm kann sein:
  - eine Liste $L = (V_1, V_2, \ldots, V_n)$ von Werten unterschiedlicher Typen;
  - oder einfach ein (großer) String I: Jede Liste L lässt sich als String über einem beliebigen "Alphabet" darstellen (z.B.: ASCII, $\{0, 1\}$, etc.)
- Ein Programm liefert als Ergebnis einen Wert (von beliebigem Typ) mittels "return" Anweisung.

Wiederholung: Unsere Programmiersprache SIMPLE

GRAPH-ERREICHBARKEIT:
INSTANZ: Ein Graph $G = (V, E)$ und Knoten $u, v \in V$.
FRAGE: Gibt es im Graph G einen Pfad von u nach v?

$S := \{u\}$;
repeat
    $S' := S$;
    for all $i \in S'$ **do** $\{$
        for all $j \in V$ **do** $\{$
            if $(i, j) \in E$ **then** $S := S \cup \{j\}$;
        $\}$
    $\}$
until $S = S'$;
if $v \in S$ **then** return *true*;
else return *false*;

Church-Turing These

Church-Turing These - Informelle Variante

Jeder Algorithmus lässt sich mittels SIMPLE realisieren.

Alternativ: Jedes “angemessene” formale Modell eines Algorithmus ist gleichmächtig (äquivalent) zur SIMPLE Programmiersprache.

Erweiterte Church-Turing These

Jedes “angemessene” formale Modell eines Algorithmus ist von seinem Zeit/Speicherbedarf äquivalent (modulo eines polynomiellen Overheads) zur SIMPLE Programmiersprache.

Wiederholung: Reduktionen

Idee 1 (nicht auf die Theoretische Informatik beschränkt)

- Angenommen, wir wollen ein neues Problem A lösen, (d.h.: wir wollen einen Algorithmus für das Problem A entwickeln).
- Und angenommen, wir wissen, wie man ein verwandtes Problem B löst (d.h.: wir haben bereits einen passenden Algorithmus für das Problem B).
- Idee. Wir könnten versuchen, A zu lösen, indem wir es zum Problem B umformen, (d.h.: wir entwickeln einen Algorithmus für Problem A, der den Algorithmus für Problem B verwendet).

Schlussfolgerung: Wenn diese Strategie funktioniert, sagen wir:
"Problem A wird auf Problem B reduziert" und wir schreiben $A \leq B$.
$\Longrightarrow$ Problem A ist mindestens so leicht lösbar ist wie Problem B.

Wiederholung: Reduktionen

Idee 2 (typisch für Berechenbarkeit und Komplexität)

- Angenommen, es gelingt uns nicht, für ein neues Problem B ein geeigtnes Lösungsverfahren zu finden, d.h.: wir finden keinen (effizienten) Algorithmus oder nicht einmal ein Semi-Entscheidungsverfahren für Problem B.
- Und angenommen, wir wissen, dass ein verwandtes Problem A *schwer* zu lösen ist, d.h.: es wurde bereits bewiesen, dass es keinen (effizienten) Algorithmus oder nicht einmal ein Semi-Entscheidungsverfahren für Problem A gibt.
- Idee. Wir entwickeln einen Algorithmus für Problem A, der einen (hypothetischen) Algorithmus für Problem B verwendet.

Schlussfolgerung: Wenn diese Strategie funktioniert, sagen wir wiederum: "Problem A wird auf Problem B reduziert" und wir schreiben $A \leq B$.
$\Longrightarrow$ Problem B ist mindestens so schwer lösbar ist wie Problem A.

Wiederholung: Reduktionen

Beschränkung der Ressourcen

- Eine Problemreduktion von A nach B sollte einfacher sein als die beiden betrachteten Probleme.
- Wenn das nicht der Fall wäre, könnte die Reduktion einen Teil der Komplexität von A beseitigen und ein Vergleich zwischen (den Schwierigkeitsgraden von) A und B wäre nicht mehr möglich.

Typische Anforderung in der Berechenbarkeitstheorie

Reduktionen müssen berechenbar sein.

Typische Anfordering in der Komplexitätstheorie

Reduktionen müssen effizient berechenbar sein, z.B. in polynomieller Zeit, d.h.: $O(n^k)$ für Instanzen der Größe n und Konstante $k \geq 1$.

Wiederholung: Reduktionen

"Many-One" Reduktionen

- Definiere eine Funktion R von der Menge der Instanzen von Problem A auf die Menge der Instanzen von Problem B, d.h.: jede Instanz x von Problem A wird auf eine Instanz $R(x)$ von Problem B abgebildet.
- Wenn man nun die Instanz $R(x)$ mit einem Lösungsverfahren für Problem B löst, dann ist das Ergebnis für die Instanz $R(x)$ bereits das korrekte Ergebnis für die Instanz x des Problems A.
- In diesem Fall sagen wir: x und $R(x)$ sind äquivalent.
  Für Entscheidungsprobleme A und B bedeutet das: x ist eine positive Instanz von $A \Leftrightarrow R(x)$ ist eine positive Instanz von B.
- Ressourcenbeschränkung: Die Funktion R muss (effizient) berechenbar sein. Im Fall einer Beschränkung auf polynomielle Zeit, heißt so eine Reduktion "Karp Reduktion".

Aussagenlogik

Syntax

Die Syntax der Aussagenlogik (d.h. die Menge der wohl-geformten aussagenlogischen Formeln) basiert auf folgenden Symbolen:

- Boole'sche Variable (oder Atome): $X = \{x_1, x_2, \ldots\}$.
- Boole'sche Konnektive: $\vee$, $\wedge$, und $\neg$.

Die Menge der aussagenlogischen Formeln ist die kleinste Menge, die folgenden Bedingungen genügt:

- alle Atome sind aussagenlogische Formeln;
- wenn ϕ_1 und ϕ_2 aussagenlogische Formeln sind, dann sind dies auch $\neg\phi_1$, $(\phi_1 \wedge \phi_2)$ und $(\phi_1 \vee \phi_2)$.

Im folgenden werden wir einfachheitshalber von Formeln und Atomen sprechen. Weiters werden wir übliche Konventionen (weglassen von Klammern, etc.) nutzen.

Aussagenlogik

- Eine Formel der Form form x_i oder $\neg x_i$ nennen wir Literal.
- Eine Disjunktion von Literalen nennen wir eine Klausel
- Eine Konjunktion von Klauseln nennen wir eine Konjunktive Normalform (KNF).
- Wenn jede Klausel aus n Literalen besteht, nennen wir eine KNF auch n-KNF.
- Eine 3-KNF Formel ist daher von der Form

$$\bigwedge_{i=1}^{m} \left(l_{i,1} \vee l_{i,2} \vee l_{i,3} \right)$$

Aussagenlogik

Wie lassen sich aussagenlogische Formeln auswerten?
Beobachtung: Formeln sind Aussagen die entweder wahr oder falsch sind und setzen sich aus Atomen zusammen, die ebenfalls entweder wahr oder falsch sein können.

Definition

Eine Wahrheitsbelegung I ist eine Funktion $I \colon X \to \{\mathbf{true}, \mathbf{false}\}$.

$I(\cdot)$ wird induktiv von Atomen auf beliebige Formeln erweitert:

- Für $\phi = \neg\phi_1$ gilt, dass $I(\phi) = \mathbf{true}$ genau dann wenn (gdw) $I(\phi_1) = \mathbf{false}$.
- Für $\phi = \phi_1 \wedge \phi_2$ gilt, dass $I(\phi) = \mathbf{true}$ gdw $I(\phi_1) = I(\phi_2) = \mathbf{true}$.
- Für $\phi = \phi_1 \vee \phi_2$ gilt dass $I(\phi) = \mathbf{true}$ gdw $I(\phi_1) = \mathbf{true}$ oder $I(\phi_2) = \mathbf{true}$.

Wenn wir von einer konkreten Formel ϕ sprechen, werden wir üblicherweise die Wahrheitsbelegung auf jene Atome beschränken, die in ϕ vorkommen. Wir sprechen dann von einer Wahrheitsbelegung für ϕ.

Einige klassische Entscheidungsprobleme der Aussagenlogik

MODEL-CHECKING

INSTANZ: Aussagenlogische Formel ϕ und Wahrheitsbelegung I für ϕ.
FRAGE: Gilt $I(\phi) = \textbf{true}$ (d.h. ist ϕ wahr unter I)?

SAT (SATISFIABILITY)

INSTANZ: Aussagenlogische Formel ϕ.
FRAGE: Ist ϕ erfüllbar? (d.h.: gibt es eine Wahrheitsbelegung I für ϕ, sodass $I(\phi) = \textbf{true}$).

3-SAT

INSTANZ: Aussagenlogische Formel ϕ in 3-KNF.

FRAGE: Ist ϕ erfüllbar?

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

k-Färbbarkeit

Sei $k \in \mathbb{N}$ mit $k \geq 2$. Dann ist k-Färbbarkeit folgendermaßen definiert:

Instanz: ungerichteter Graph $G = (V, E)$
Frage: Ist der Graph G k-färbbar?
d.h.: gibt es eine Funktion $f: V \to \{0, \ldots, k-1\}$, so dass für alle Kanten $[v_i, v_j] \in E$ gilt: $f(v_i) \neq f(v_j)$.

3-COL (Drei-Färbbarkeit)

Instanz: ungerichteter Graph $G = (V, E)$

Frage: Ist G drei-färbbar? Das heißt, können wir jedem Knoten aus V eine von 3 Farben so zuordnen, dass alle adjazenten Knoten verschieden gefärbt sind?

Anm: Im folgenden verwenden wir als Farben $\{r, g, b\}$ anstatt $\{0, 1, 2\}$.

Wiederholung: Vorgangsweise bei Korrektheitsbeweisen

- Zur Erinnerung: eine Many-One Reduktion R von Problem A auf Problem B ist korrekt, wenn für jede beliebige Instanz x von A die Äquivalenz gilt: x ist eine positive Instanz von $A \Leftrightarrow R(x)$ ist eine positive Instanz von B.
- Üblicherweise zeigt man die 2 Richtungen "$\Rightarrow$" und "$\Leftarrow$" der Äquivalenz getrennt.
- Der Beweis beginnt immer mit der Annahme "Sei x eine positive Instanz von A." bzw. "Sei $R(x)$ eine positive Instanz von B.".
- Mittels *schlüssiger* Beweisschritte versucht man zu zeigen, dass man dann auch beim anderen Problem eine positive Instanz hat.
- Typische Beweisschritte sind, wenn etwas *aufgrund der Annahme*, *aufgrund einer Definition* (z.B. einer "Lösung") oder *aufgrund der Problemreduktion* gilt.
- Es erhöht die Lesbarkeit eines Beweises, wenn man im Beweis klarstellt, was man gerade möchte bzw. was noch zu zeigen ist.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

Problemreduktion

Sei $G = (V, E)$ eine beliebige Instanz von 3-COL, d.h.: $G = (V, E)$ ist ein ungerichteter Graph mit Knoten $V = \{v_1, \ldots, v_n\}$ und Kanten E.

Für die zu konstruierende Formel verwenden wir Atome a_i^c, mit $1 \leq i \leq n$ und $c \in \{r, g, b\}$. Intuition: Wenn a_i^c auf wahr gesetzt ist, dann ist Knoten v_i mit Farbe c gefärbt.

Wir definieren Reduktion $R(\cdot)$, die für jeden Graph G, eine SAT-Instanz $R(G) = \phi_G$ liefert (in polynomieller Zeit), wobei $\phi_G = \phi_1 \wedge \phi_2 \wedge \phi_3$ mit

$$\phi_1 = \bigwedge_{1 \leq i \leq n} (a_i^r \vee a_i^g \vee a_i^b),$$

$$\phi_2 = \bigwedge_{1 \leq i \leq n} \left(\neg(a_i^r \wedge a_i^g) \wedge \neg(a_i^r \wedge a_i^b) \wedge \neg(a_i^g \wedge a_i^b)\right),$$

$$\phi_3 = \bigwedge_{[v_i, v_j] \in E} \left(\neg(a_i^r \wedge a_j^r) \wedge \neg(a_i^g \wedge a_j^g) \wedge \neg(a_i^b \wedge a_j^b)\right).$$

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

Wir müssen folgende Äquivalenz zeigen:
$G = (V, E)$ ist eine positive Instanz von 3-COL $\Leftrightarrow$
ϕ_G ist eine positive Instanz von SAT

$\Rightarrow$: Angenommen $G = (V, E)$ ist eine positive Instanz von 3-COL. Dann gibt es eine Funktion $f: V \to \{r, g, b\}$, die eine gültige 3-Färbung für den Graph $G = (V, E)$ ist, d.h.: für jede Kante $[v_i, v_j]$ in G gilt $f(v_i) \neq f(v_j)$.
Wir konstruieren eine Wahrheitsbelegung I für ϕ_G:

$$I(a_i^c) = \begin{cases} \mathit{true} & \text{if } f(v_i) = c, \\ \mathit{false} & \text{if } f(v_i) \neq c. \end{cases}$$

und zeigen, dass $I(\phi_G) = \textbf{true}$.
Da $\phi_G = \phi_1 \wedge \phi_2 \wedge \phi_3$, müssen wir sowohl $I(\phi_1) = \textbf{true}$, $I(\phi_2) = \textbf{true}$, als auch $I(\phi_3) = \textbf{true}$ zeigen.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

$I(\phi_1) = \textbf{true}$: Betrachte beliebige Klausel $(a_i^r \vee a_i^g \vee a_i^b)$ aus ϕ_1.

Laut Annahme ist f eine gültige Dreifärbung von G. Also ist $f(v_i) \in \{r, g, b\}$.

Laut Definition von I gilt dann für mindestens eine der Atome (a_i^r, a_i^g, a_i^b), dass sie in I den Wahrheitswert true hat.

Daher erfüllt I die Klausel $(a_i^r \vee a_i^g \vee a_i^b)$.

Da I jede Klausel von ϕ_1 erfüllt, haben wir gezeigt, dass $I(\phi_1) = \textbf{true}$.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

$I(\phi_2) = \textbf{true}$: Betrachte beliebiges Konjunkt

$$c_i = \neg(a_i^r \wedge a_i^g) \wedge \neg(a_i^r \wedge a_i^b) \wedge \neg(a_i^g \wedge a_i^b)$$

aus ϕ_2 $(1 \leq i \leq n)$.

Laut Annahme ist f eine gültige Dreifärbung von G. Also ist $f(v_i) \in \{r, g, b\}$.

Laut Definition von I gilt dann für mindestens zwei der drei Atome (a_i^r, a_i^g, a_i^b), dass sie in I den Wahrheitswert false hat.

Daher sind sowohl $(a_i^r \wedge a_i^g)$, $(a_i^r \wedge a_i^b)$, als auch $(a_i^g \wedge a_i^b)$ falsch unter I. Es folgt, dass $I(c_i) = \textbf{true}$.

Da I jedes solche Konjunkt von ϕ_2 erfüllt, haben wir gezeigt, dass $I(\phi_2) = \textbf{true}$.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

$I(\phi_3) = $ **true**: Zur Erinnerung ...

$$\phi_3 = \bigwedge_{[v_i,v_j]\in E} \left(\neg(a_i^r \wedge a_j^r) \wedge \neg(a_i^g \wedge a_j^g) \wedge \neg(a_i^b \wedge a_j^b)\right).$$

Wir beweisen diesen Teil indirekt, also angenommen $I(\phi_3) = $ **false**.
Dann ist zumindest eines der Konjunkte

$$c_{i,j} = \left(\neg(a_i^r \wedge a_j^r) \wedge \neg(a_i^g \wedge a_j^g) \wedge \neg(a_i^b \wedge a_j^b)\right)$$

falsch in I.

Es folgt, dass $(a_i^r \wedge a_j^r)$, $(a_i^g \wedge a_j^g)$, oder $(a_i^b \wedge a_j^b)$ wahr unter I.
Daher $I(a_i^c) = I(a_j^c) = 1$ für eine der drei Farben $c \in \{r, g, b\}$.

Laut Definition von I haben wir auch $f(v_i) = f(v_j) = c$.
Laut Problemreduktion enthält ϕ_3 Konjunkt $c_{i,j}$ nur dann, wenn es eine Kante $[v_i, v_j]$ in G gibt. Laut Annahme ist f eine gültige Dreifärbung von G, also $f(v_i) \neq f(v_j)$. Widerspruch!

Wir haben gezeigt, dass $I(\phi_G) = $ **true**. Daher ist ϕ_G eine positive Instanz von SAT.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

$\Leftarrow$: Angenommen ϕ_G ist eine positive Instanz von SAT, gegeben Graph $G = (V, E)$. Dann gibt es eine Wahrheitsbelegung I, sodass $I(\phi_G) = \mathbf{true}$.
Zuerst zeigt man dass aus $I(\phi_1) = \mathbf{true}$ und $I(\phi_2) = \mathbf{true}$ folgt: für jedes i gilt, dass genau eines der Atome aus (a_i^r, a_i^g, a_i^b) unter I den Wahrheitswert true hat.
Wir definieren nun eine Funktion $f : V \to \{r, g, b\}$ sodass, für alle $1 \leq i \leq n$, $f(v_i) = c$, wobei $c \in \{r, g, b\}$ genau jenes c mit $I(a_i^c) = \mathbf{true}$ ist.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

Fortsetzung der $\Leftarrow$-Richtung:
Wir müssen noch zeigen, dass f eine gültige Drei-Färbung von G ist. Wir zeigen das abermals indirekt und nehmen an, dass es eine Kante $[v_i, v_j]$ in G gibt, sodass $f(v_i) = f(v_j) = c$.
Laut Definition von f, gilt daher dass $I(a_i^c) = I(a_j^c) = \textbf{true}$.
Laut Problemreduktion enthält ϕ_3 dann die Teilformel $\neg(a_i^c \wedge a_j^c)$. Es folgt, dass $I(\neg(a_i^c \wedge a_j^c)) = \textbf{false}$ und daher $I(\phi_G) = \textbf{false}$.
Laut Annahme ist aber $I(\phi_G) = \textbf{true}$. Widerspruch.
Es gilt daher, dass f eine gültige Drei-Färbung von G ist: somit ist G eine positive Instanz von 3-COL.

Teil 3: Einführung in Komplexitätstheorie

Teil 3.1: Grundlegende Konzepte und Wiederholung

Teil 3.2: Die Komplexitätsklassen P und NP

Teil 3.3: NP-Vollständigkeit

Teil 3.4: Weitere Komplexitätsklassen

Prolog

Komplexitätsanalyse

- Klassifiziert konkrete Probleme anhand deren Schwierigkeit
- Dies geschieht typischerweise mittels Komplexitätsklassen

Komplexitätstheorie

- Untersucht Eigenschaften von Komplexitätsklassen
- ... und Beziehungen zwischen Komplexitätsklassen

Wir verstehen unter Komplexität hier immer die Worst-Case Komplexität. Wir gehen nicht auf andere Aspekte ein, z.B. average-case Komplexität, best-case Komplexität, Komplexität für "Instanzen, die in der Praxis auftreten" ...

Laufzeit und Speicherbedarf

Wir messen die die Effizienz von Algorithmen bzw. die Komplexität von Problemen in Bezug auf (Lauf)zeit und Speicher(bedarf):

Laufzeit

Gegeben Programm Π und Input I für Π, die Laufzeit von Π auf I ist die Anzahl von atomaren Operationen die für die Ausführung von Π auf I benötigt wird.

- Der Term "Atomare Operation" hängt von der Programmiersprache, bzw. der Computer Architektur ab.

Speicherbedarf

Gegeben Programm Π und Input I für Π, der Speicherbedarf von Π auf I ist die Anzahl der Bits die für die Ausführung von Π auf I im Speicher benötigt wird.

Komplexität eines Algorithmus

Messen von Komplexität

Wir basieren unsere Überlegungen auf die asymptotische, worst-case Komplexität eines Algorithmus, d.h. anhand einer Funktion $f(\,)$, sodass für jede Probleminstanz der Größe n, die Antwort in $\mathrm{O}(f(n))$ Operationen (Laufzeit), bzw. mittels $\mathrm{O}(f(n))$ Bits im Speicher (Speicherbedarf) berechnet werden kann.

Recall

Seien f, g: $\mathbf{N} \mapsto \mathbf{N}$.

- $g(n) = \mathrm{O}(f(n))$ (g wächst maximal wie f), wenn es c und n_0 gibt, sodass für alle $n \geq n_0$, $g(n) \leq c \cdot f(n)$.

Anmerkung: Da wir Laufzeit in Bezug auf die Instanzgröße messen, ist eine "vernünftige" Repräsentation der Instanz wichtig.
Vergleiche unäre vs. binäre oder dezimale Zahlendarstellung.

Komplexität eines (Entscheidungs)Problems

Komplexität eines Problems =
Worst-case Komplexität des bestmöglichen Algorithmus für dieses Problem.

Zentrale Unterscheidung:

- Tractable Problems - es gibt einen effizienten Algorithmus
- Intractable Problems - es gibt (vermutlich) keinen effizienten Algorithmus

Positive Resulte sind daher “einfach” zu zeigen.
Für negative Resulte brauchen wir formale Konzepte.

Vergleiche entscheidbare/unentscheidbare Probleme.

Nochmal: Erweiterte Church-Turing These

Frage

Beeinflußt das konkrete Berechnungsmodell (die konkrete Programmiersprache) oder eine spezifische Architektur die Zeit/Speicher-Performanz eines implementierten Algorithmus?

- Wir können dies i.A. vernachlässigen: die Auswahl einer spezifischen Programmiersprache/Architektur bringt nur eine geringfügige Verbesserung in Bezug auf Laufzeit/Speicherbedarf.

  Ein effizientes Programm kann in jede andere Programmiersprache (z.B. SIMPLE) übersetzt werden ohne dass Effizienz verloren geht.

- "effizient" bedeutet hier: polynomiell
- Daher können wir für unsere Analysen in Bezug auf Existenz/Nicht-Existenz von effizienten Algorithmen für ein konkretes Problem auf SIMPLE als Berechnungsmodell basieren.

Die Komplexitätsklasse P

Die Klasse P ist die Menge aller Entscheidungsprobleme die in polynomieller Zeit (in Bezug auf die Instanzgröße) gelöst werden können.

Notation: $|I|$ ist die Größe einer Instanz I.

Definition

Die Klasse P beinhaltet genau jene Entscheidungsprobleme $\mathcal{P}$ die folgende Eigenschaften haben:

1. es gibt ein SIMPLE Programm Π für $\mathcal{P}$, sodass
2. für alle Instanzen I von $\mathcal{P}$ die Laufzeit von Π auf I polynomiell in $|I|$ ist, d.h. die Laufzeit ist durch $\mathrm{O}(|I|^k)$, wobei k eine Konstante, beschränkt.

Die Komplexitätsklasse P

Zahlreiche wichtige Entscheidungsprobleme sind in P:

- Graph-Erreichbarkeit ist in P;
- 2-SAT ist in P (recall Block1: Reduktion von 2-SAT auf Graph-Erreichbarkeit);
- Testen ob ein String aus einer kontextfreien Grammatik (recall Block2) abgeleitet werden kann, ist in P.
- Diskussion: Primzahlentest?

Entscheidungsprobleme aus P

Boolean Circuits

Ein Boolean Circuit der Arität k ist ein gerichteter azyklischer Graph (DAG) bestehend aus: k Input gates $in_1, \ldots, in_k$, einem Output gate, und einer beliebigen Anzahl von *AND*, *OR* und *NOT* gates.
Input gates haben keine Kind-gates.
NOT gates und das Output gate haben genau ein Kind-gate,
AND bzw. OR gates haben zumindest zwei Kind-gates.

CIRCUIT-EVAL

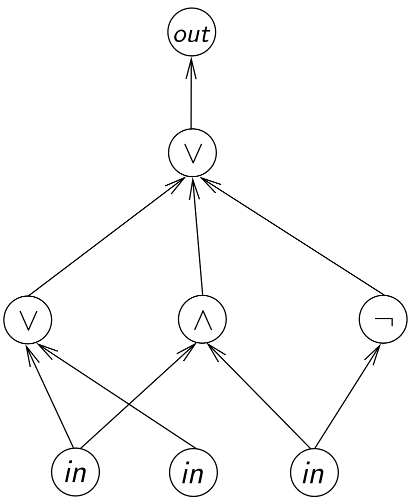
INSTANZ: Ein Boolean Circuit C und eine Funktion I die jedem Input gate von C einen Wert **true** oder **false** zuweist.

FRAGE: Ist der output von C **true** unter Belegung I?

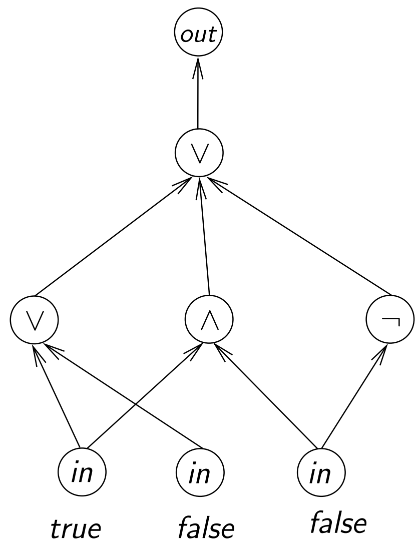
Belegungen werden im Circuit folgendermaßen propagiert:

- *AND* gate: Output **true** gdw alle Kind-gates Output **true**,
- *OR* gate: Output **true** gdw zumindest ein Kind-gate Output **true**,
- *NOT* gate: Output **true** gdw wenn das Kind-gate Output **false**.

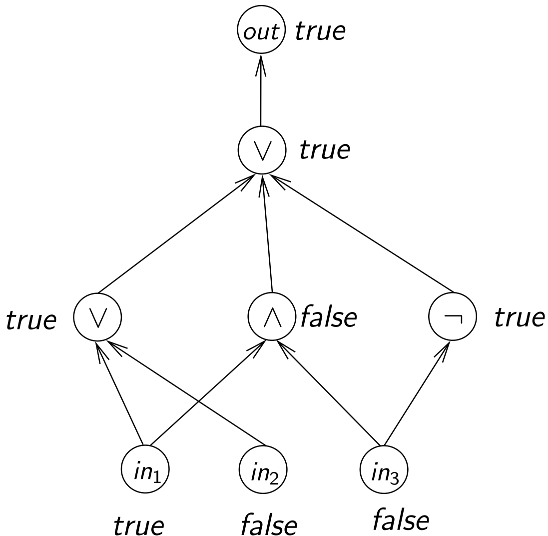
Beispiel für einen Circuit



Beispiel für einen Circuit mit Belegung



Beispiel für einen ausgewerteten Circuit



CIRCUIT-EVAL ist in P

Algorithmus **evaluateCircuit**

Input: Boolean Circuit C der Arität k, Belegung I für C
Output: **true** gdw Output von C ist **true** in I

1. Setze $A := \{(in_1, I(in_1)), \ldots, (in_k, I(in_k))\}$; /* Kopie von I */
2. Sei G die Menge aller Gates in C;
   /* jene $g \in G$ die in A aufscheinen nennen wir value-assigned */
3. **while** existiert $g \in G$ sodass g nicht value-assigned **do**
   (i) wähle ein $g \in G$ das nicht value-assigned, dessen Kind gates aber allle bereits value-assigned sind (ein solches Gate muss existieren da C ein DAG mit den Input gates als Quellknoten)
   (ii) gib zu A das Paar (g, v), wobei v der Wert der sich durch den Typ von g und der Werte der Kind gates von g gemäß A ergibt.
4. **if** $(out, true) \in A$ **then return** *true* **else return** *false*

CIRCUIT-EVAL ist in P

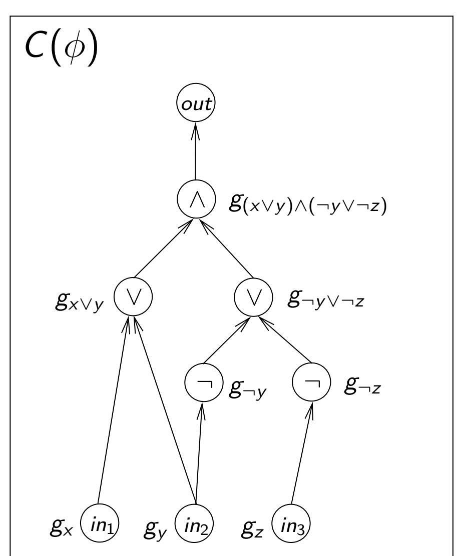
Wir argumentieren dass Algorithmus **evaluateCircuit** in polynomieller Zeit (in Bezug auf Instanzgröße) läuft:

1. Die Relation A ist zu jeder Zeit linear in der Größe von C.
2. Da jede Iteration der Schritte (i-ii) ein neues Gate belegt, haben wir maximal $|G|$ Iterationen des **while** Loops.
3. Gegeben Gate $g \in G$, um zu testen, dass g unbelegt und alle Kind-Gates belegt, kann in quadratischer Zeit (in Bezug auf die Größe von G) bewerkstelligt werden.
4. Aufgrund von (3) hat Schritt (i) kubische Laufzeit (durchsuche G und führe den Test des vorigen Punkts aus).
5. Schritt (ii) ist linear in der Größe von C.
6. Die Laufzeit des Algorithmus ist daher mit $\mathrm{O}(n^4)$ beschränkt.

Tatsächlich gibt es für dieses Problem einen $\mathrm{O}(n)$-Algorithmus.

Aussagenlogische Formulen und Boolean Circuits

Example:
$\phi = (x \vee y) \wedge (\neg y \vee \neg z)$



Model Checking ist in P

MODEL-CHECKING

INSTANZ: Aussagenlogische Formel ϕ und Wahrheitsbelegung I für ϕ.

FRAGE: $I(\phi) = \textbf{true}$?

Wir können nun folgern, dass **MODEL-CHECKING** in P.

- Wir reduzieren **MODEL-CHECKING** auf **CIRCUIT-EVAL**. Die Reduktion selbst ist klarerweise in polynomieller Zeit möglich.
- Weiters haben wir bereits argumentiert, dass **CIRCUIT-EVAL** in P.

Diskussion

- Ein Problem gilt als effizient lösbar (tractable) wenn es in P ist. Zur Erinnerung: das bedeutet, dass es einen Algorithmus für dieses Problem gibt, sodass die Lösungszeit für jede Instanz des Problems polynomiell zu dessen Größe n steht, also die Laufzeit durch $\mathrm{O}(n^d)$ beschränkt ist. (daher: asymptotic, worst-case performance).
- In der Praxis kann es vorkommen, dass dieses Konzept nur bedingt greift. Beispiel: Linear Programming.
  - Es existieren Algorithmen mit polynomieller worst-case Komplexität.
  - In der Praxis hat sich allerdings die Simplex Methode (exponentielle worst-case Komplexität) als brauchbarer herauskristallisiert.
- Wenn es für ein Problem keinen Polynomialzeit-Algorithmus gibt, nennen wir ein solches Entscheidungsproblem intractable.
  - Intractability (kein effizienter Algorithmus bekannt, Existenz eines solchen Algorithmus unwahrscheinlich, aber nicht auszuschliessen) $\longrightarrow$ Komplexitätsklasse NP ... folgt in Kürze.
  - beweisbare Intractability (i.e., Existenz eines effizienten Algorithmus kann formal ausgeschlossen werden) $\longrightarrow$ Komplexitätsklasse EXPTIME.

Die Komplexitätsklasse NP

Die Komplexitätsklasse NP

- Gegeben Formel ϕ und Wahrheitswertwertbelegung I: zu überprüfen, ob $I(\phi) = \textbf{true}$ ist effizient lösbar.
  - Recall: **MODEL-CHECKING** ist in P.
- Um Erfüllbarkeit (SAT) von ϕ zu entscheiden, könnten wir alle möglichen Wahrheitswertbelegungen für ϕ durchgehen und jeweils effizient prüfen ob die Belegung ϕ erfüllt.
- Naives Aufzählen aller Wahrheitswertbelegungen für ϕ braucht allerdings exponentielle Laufzeit:
  - 2^n mögliche Wahrheitswertbelegungen, wenn n die Anzahl der Atome in ϕ.
- Gibt es eine schlauere Strategie? Hard to say...
- Mit Glück finden wir die richtige Belegung möglichst bald und unser Algorithmus für SAT könnte sofort terminieren. (aber das hilft nicht bei negativen Instanzen).
- "Mit Glück" kann durch Nicht-Determinismus erzwungen werden. Tatsächlich steht NP für Non-deterministic Polynomial Time.

Die Komplexitätsklasse NP

Viele Probleme lassen sich durch sogenannte Zertifikate (auch Zeugen, Lösungen, etc) analysieren. Am Beispiel SAT:

- Wenn ϕ erfüllbar gibt es zumindest eine Wahrheitswertbelegung für ϕ, die ϕ wahr macht.
- Für unerfüllbare Formeln existiert keine solche Belegung.

Beobachtung 1: Im Falle von SAT sind alle Zertifikate kompakt ... eine Wahrheitswertbelegung für Formel ϕ kannn durch einen String repräsentiert werden der polynomiell (linear) in der Größe von ϕ ist.

Beobachtung 2: Überprüfen ob es sich bei einer Belegung tatsächlich um ein Zertifikat für eine Instanz handelt, ist effizient lösbar.

Die Komplexitätsklasse NP

SAT ist nur eines von vielen Problemen die der Komplexitätsklasse **NP** zugeschrieben werden.

Informell ist NP die Menge jener Entscheidungsprobleme $\mathcal{P}$ für die gilt:

- positive Instanzen von $\mathcal{P}$ haben kompakte Zertifikate
- gegeben eine Instanz I von $\mathcal{P}$ und Zertifikat-Kandidat C, dann ist es einfach zu entscheiden ob C als Zertifikat von I gilt.

Recall: kompakt und "einfach zu entscheiden" beziehen sich auf die Instanzgröße.

Zertifikat-Relation

Wir definieren zuerst das Konzept der Zertifikat-Relation.

Definition (Zertifikat-Relation)

Sei $\mathcal{P}$ ein Eintscheidungsproblem, $INSTANCES(\mathcal{P})$ die Menge der Instanzen von $\mathcal{P}$.

Eine Zertifikat-Relation für $\mathcal{P}$ ist eine Relation $R \subseteq INSTANCES(\mathcal{P}) \times STRINGS$ (wobei $STRINGS$ die Menge aller Zeichenketten ist), sodass

$I \in INSTANCES(\mathcal{P})$ ist eine positive Instanz von $\mathcal{P}$

gdw

existiert Zertifikat $C \in STRING$ sodass $(I, C) \in R$.

Die Komplexitätsklasse NP

Wir können nun unsere Intuition bzgl. kompakte und einfach zu entscheidene Zertifikate formalisieren.

Definition

Gegeben eine binäre Relation R.

R heißt polynomiell balanziert wenn für alle $(v_1, v_2) \in R$, $|v_2| \leq |v_1|^k$ (für fixes $k \geq 1$) gilt.

R heißt polynomiell entscheidbar wenn es einen Algorithmus gibt, welcher, gegeben v_1 und v_2, $(v_1, v_2) \in R$ in polynomieller Zeit (in der Größe von v_1, v_2) prüft.

Definition (Die Komplexitätsklasse NP)

Ein Entscheidungsproblem $\mathcal{P}$ ist in der Klasse NP wenn es eine polynomiell balanzierte und polynomiell entscheidbare Zertifikats-Relation für $\mathcal{P}$ gibt.

Beispiel: **SAT** ist in **NP**

Theorem

SAT $\in$ **NP**.

Beweis.

Um **SAT** $\in$ **NP** zu beweisen, müssen wir lediglich eine Zertifikats-Relation für **SAT** angeben, die auch polynomiell balanziert und polynomiell entscheidbar ist. Betrachte

$$R = \{(\phi, I) \mid \text{Formel } \phi \text{ ist } \textbf{true} \text{ unter Belegung } I\}.$$

- R ist eine Zertifikats-Relation für **SAT**: ϕ ist eine positive Instanz von **SAT** $\Leftrightarrow$ es gibt eine Wahrheitswertbelegung I die ϕ wahr macht $\Leftrightarrow$ $(\phi, I) \in R$.
- R is polynomiell balanziert: jede Belegung I für ϕ kann einfach als Teilmenge von Atomen in ϕ repräsentiert werden.
- R is polynomiell entscheidbar: **MODEL-CHECKING** ist in P!

□

Die Komplexitätsklasse NP - Alternative Definition

Wir erweitern unsere SIMPLE programming language um einen Befehl `choice`(*assignment1*,*assignment2*). Ein `choice` Befehl teilt die Berechnung in zwei Zweige auf (die dann parallel und unabhängig voneinander weiterlaufen), wobei in einem Zweig das *assignment1*, im anderen das *assignment2* gesetzt wird.
Ein SIMPLE program mit `choice` Befehlen und Boole'schen Return-Values nennen wir `choice` Programm.

- Die Laufzeit eines `choice` Programms wird als die Laufzeit des längsten Berechungszweigs definiert.
- Ein `choice` Programm liefert true, wenn zumindest ein Berechnungszweig true liefert.

Die Komplexitätsklasse NP - Alternative Definition

SAT with extended SIMPLE

Boolean test(String s, Integer n) /* s a formula over atoms $a_1, \ldots, a_n$ */
    for all $1 \leq i \leq n$ **do** {
        choice($t_i = true$,$t_i = false$);
    }
    return modelcheck (s,[$t_1, \ldots, t_n$]);   /* checks if s is true under
            the assignment mapping a_i to t_i ($1 \leq i \leq n$) */

Die Komplexitätsklasse NP - Alternative Definition

Wir können nun eine alternative und äquivalente Definition der Klasse NP (die der Definition von P ähnelt) angeben:

Definition

Die Klasse NP beinhaltet genau jene Entscheidungsprobleme $\mathcal{P}$ die folgende Eigenschaften haben:

1. es gibt ein choice Programm Π für $\mathcal{P}$, sodass
2. für alle Instanzen I von $\mathcal{P}$ die Laufzeit von Π auf I polynomiell in $|I|$ ist, d.h. die Laufzeit ist durch $\mathrm{O}(|I|^k)$, wobei k eine Konstante, beschränkt.

Beobachtung 1: Jedes Problem in P ist auch in NP.
Beobachtung 2: Mittels choice Programmen können wir in polynomiell vielen Schritten exponentiell viele Berechnungszweige "generieren".

Frage: Ist das ein "angemessenes" Berechnungsmodell?

NP Probleme als Guess & Check Prozeduren

- Jedes Problem in NP kann man als Guess & Check Prozedur (mittels `choice` Programm) effizient implementieren.
- Sei $\mathcal{P} \in \mathrm{NP}$ und R eine polynomiell balanzierte und polynomiell entscheidbare Zertifikats-Relation für $\mathcal{P}$.
- I ist eine positive Instanz von $\mathcal{P}$ wenn wir ein korrektes Zertifikat für I erraten können (guess). Das geschieht mit `choice` statements.
- Da R polynomiell balanziert ist, ist jeder Kandidat für ein Zertifikat für I polynomiell in der Größe von I
  (wir brauchen nur polynomiell viele `choice`-statements)
- Da R polynomiell entscheidbar, kann ein Kandidat als korrektes Zertfikat in polynomieller Zeit identifiziert werden (check).
  .... mittels standard SIMPLE Befehlen.

3 Wege um NP-Membership zu beweisen

3-COL (Drei-Färbbarkeit)

Instanz: ungerichteter Graph $G = (V, E)$

Frage: Ist G drei-färbbar? Das heißt, können wir jedem Knoten aus V eine von 3 Farben so zuordnen, dass alle adjazenten Knoten verschieden gefärbt sind?

1. Reduktion auf Problem in NP (siehe Reduktion 3-COL auf SAT);
2. Zertifikat-Relation:

$$R = \{((V, E), f) \mid f(v_i) \neq f(v_j) \text{ für alle } v_i, v_j \in V \text{ mit } [v_i, v_j] \in E\}$$

3. Extended SIMPLE program (Guess and Check Prozedur)
   - Rate für jeden Knoten eine Farbe (mittel zwei choice statements).
   - Überprüfe ob diese Belegung eine gültige Färbung der Knoten erzeugt.

Teil 3: Einführung in Komplexitätstheorie

Teil 3.1: Grundlegende Konzepte und Wiederholung

Teil 3.2: Die Komplexitätsklassen P und NP

Teil 3.3: NP-Vollständigkeit

Teil 3.4: Weitere Komplexitätsklassen

Komplexität: Härte und Vollständigkeit

Konvention

- Im folgenden werden wir, wenn nicht anders angegeben, polynomielle many-one Reduktionen nutzen. Wir schreiben $\mathcal{P} \leq_{\mathrm{R}} \mathcal{P}'$ wenn Problem $\mathcal{P}$ auf $\mathcal{P}'$ auf diese Art reduziert werden kann.
- Beachte dass die Relation $\leq_{\mathrm{R}}$ Probleme nach deren Schwierigkeit ordnet. Klarerweise ist $\leq_{\mathrm{R}}$ reflexiv und transitiv.

Komplexität: Härte und Vollständigkeit

Definition

Sei C eine Komplexitätsklasse und sei $\mathcal{P}$ ein Entscheidungsproblem.

$\mathcal{P}$ heißt C-hart (C-schwer) wenn jedes Problem $\mathcal{P}' \in \mathrm{C}$ auf $\mathcal{P}$ reduziert werden kann.

$\mathcal{P}$ heißt C-vollständig wenn $\mathcal{P}$ in C liegt und $\mathcal{P}$ C-hart ist.

In anderen Worten:

Vollständigkeit $=$ Mitgliedschaft und Härte

Beobachtung

Für alle Probleme $\mathcal{P}$ und $\mathcal{P}^*$ gilt: wenn $\mathcal{P}$ C-hart und $\mathcal{P} \leq_{\mathrm{R}} \mathcal{P}^*$ dann ist auch $\mathcal{P}^*$ C-hart.

Die Rolle von Vollständigkeit in der Komplexitätstheorie

- Vollständige Probleme stellen die maximalen Elemente ihrer Klasse in Bezug auf die Reduktions-Relation $\leq_{\mathrm{R}}$ dar.
- Vollständige Probleme sind ein zentrales Konzept und Werkzeug in der Komplexitätstheorie:
  - Die Komplexität eines Problems wird durch das Zeigen der Vollständigkeit für eine Komplexitätsklasse kategorisiert.
  - Andererseits ist die Essenz einer Komplexitätsklasse durch ihre vollständigen Probleme charakterisiert.
- Vollständigkeit dient als Basis für "negative" Komplexitätsresultate: Vollständige Probleme einer Klasse C sind jene Probleme dieser Klasse für die es am wenigsten wahrscheinlich ist, dass sie in einer "schwächeren Klasse" $\mathrm{C}' \subseteq \mathrm{C}$ liegen.

Die Rolle von Vollständigkeit in der Komplexitätstheorie

Theorem

Gilt für ein NP-*vollständiges Problem* $\mathcal{P}$ *dass es in* P *liegt, dann ist* NP $=$ P.

Beweis

Wir wissen P $\subseteq$ NP (siehe Definition). Wir zeigen NP $\subseteq$ P:
Sei $\mathcal{P}'$ ein beliebiges Problem in NP.
Da $\mathcal{P}$ NP-vollständig ist, existiert eine Reduktion R von $\mathcal{P}'$ auf $\mathcal{P}$. Dann können wir jedoch sofort einen (deterministischen) Polynomialzeit-Algorithmus für $\mathcal{P}'$ konstruieren: Wandle Instanz x des Problems $\mathcal{P}'$ in $R(x)$ um; entscheide $R(x)$ mit der Entscheidungsprozedur für $\mathcal{P}$. Da wir eine polynomielle Reduktion $\mathcal{P}' \leq_{\mathrm{R}} \mathcal{P}$ und $\mathcal{P} \in$ P haben, folgt dass $\mathcal{P}' \in$ P.
Wir habem somit NP $\subseteq$ P und folgerichtig auch NP $=$ P.

Remark: Es gibt einen relativ großen Konsens dass NP $\neq$ P. In diesem Fall gibt es keine effizienten Algorithmen für NP-vollständige Probleme!

Wiederholung: klassische Entscheidungsprobleme der Aussagenlogik

SAT (Satisfiability)

Instanz: Aussagenlogische Formel ϕ.
Frage: Ist ϕ erfüllbar? (d.h.: gibt es eine Wahrheitsbelegung I für ϕ, sodass $I(\phi) = \textbf{true}$).

3-SAT

Instanz: Aussagenlogische Formel ϕ in 3-KNF.

Frage: Ist ϕ erfüllbar?

2-SAT

Instanz: Aussagenlogische Formel ϕ in 2-KNF.

Frage: Ist ϕ erfüllbar?

Enscheidungsprobleme der Aussagenlogik

- Wir haben bereits argumentiert dass 2-SAT tractable ist.
- SAT und 3-SAT sind aber intractable (Wir geben hier nur einen Sketch der Beweisidee der NP-Vollständigkeit für diese zwei Probleme).

Wichtige Beobachtung

Das SAT Problem gilt als *das* klassische NP-vollständige Problem. Es bleibt NP-vollständig wenn wir die Instanzen auf 3-KNF einschränken (3-SAT). 3-SAT ist i.A. das Problem der Wahl wenn wir für weitere Probleme NP-Vollständigkeit zeigen wollen (da dies das Finden bzw. die Definition der Reduktionen einfacher macht).

Komplexität von SAT und $3\text{-}\mathrm{SAT}$

Cook-Levin Theorem

SAT ist NP-vollständig.

Theorem

$3\text{-}\mathrm{SAT}$ ist NP-vollständig.

NP-membership

Wir haben bereits besprochen, dass sowohl SAT als auch $3\text{-}\mathrm{SAT}$ mittels eines einfachen NP-Algorithmus entschieden werden können:
Gegeben Formel ϕ,

1. rate Wahrheitswertbelegung *I* für die Atome in ϕ.
2. prüfe ob ϕ auf **true** unter *I* evaluiert.

Beweisidee NP-Härte von SAT

Um zu zeigen dass SAT NP-hart ist, müssen wir für alle Probleme $\mathcal{P}$ in NP eine polynomielle many-one Reduktion von $\mathcal{P}$ nach SAT finden.

Da NP unendlich viele Probleme beinhaltet, können wir nicht jedes Problem in NP individuell behandeln. Wir brauchen eine Methode die alle diese Probleme umfasst $\Longrightarrow$ Berechnungsmodell nutzen.

Idee. Gegeben ein beliebiges Problem $\mathcal{P} \in$ NP und eine beliebige Instanz I von $\mathcal{P}$. Wir müssen zeigen, dass wir in polynomieller Zeit eine Formel ϕ bilden können, sodass I ist eine positive Instanz von $\mathcal{P} \Leftrightarrow \phi$ ist erfüllbar.

Da $\mathcal{P} \in$ NP gibt es eine polynomiell balanzierte und polynomiell entscheidbare Zertifikats-Relation R für $\mathcal{P}$.
Recall: I ist eine positive Instanz von $\mathcal{P} \Leftrightarrow (I, C) \in R$ mit (i) Größe von C polynomiell in $|I|$, und (ii) es gibt einen Polynomialzeit-Algorithmus A um zu testen ob $(I, C) \in R$.

Beweisidee NP-Härte von SAT

Um die Reduktion zu definieren brauchen wir daher für jedes $\mathcal{P}$ und jede Instanz I eine Formel ϕ sodass Folgendes gilt:
ϕ ist erfüllbar $\Leftrightarrow$ es gibt ein polynomial-size Objekt C sodass Algorithmus A Returnwert *true* für (I, C) liefert.

Informell muss die gesuchte Formel ϕ zwei Aufgaben kodieren:

1. Zertifikat-Kandidat generieren
2. die Berechnungen von A auf dem Paar (I, C) simulieren.

Der zweite Teil ist schwierig: wie können wir eine Formel angeben die SIMPLE Programme simuliert? Wie sollen wir **while** loops, **for** loops, **if**/**then**/**else** statements kodieren?

Antwort: Im Prinzip geht das, aber die bessere Wahl ist natürlich ein einfacheres Berechnungsmodell zu verwenden. Turing Maschinen sind ein wesentlich einfacheres Modell, aber äquivalent - erweiterte Church-Turing These - und lassen sich einfacher in Aussagenlogik kodieren.

Komplexität von SAT und 3-SAT

Beweisidee NP-Härte von 3-SAT

Wir brauchen eine Polynomialzeit Reduktion von SAT nach 3-SAT, d.h. für eine beliebige Formel ϕ existiert eine 3-KNF Formel $R(\phi) = \psi$, sodass ϕ erfüllbar $\Leftrightarrow$ ψ erfüllbar.

Anmerkungen:

- Das ist nicht komplett trivial. Beachte: die Standardumwandlung in KNF mittels de Morgan'sche Regeln und Distributiv-Regeln von $\wedge$ und $\vee$ führt im allgemeinen zu einem Exponential Blow-up. (Betrachte KNFs die logisch äquivalent zu $(x_1 \wedge y_1) \vee \ldots \vee (x_n \wedge y_n)$ sind.)
- Jede Formel ϕ kann allerdings in in Polynomialzeit in eine sat-äquivalente Formel ψ in 3-KNF überführt werden
- Grundidee: Dies geschieht durch zusätzliche Atome die für Teilformeln von ϕ stehen (Tseitin-Transformation).
- ϕ und ψ sind daher i.A. nicht logisch äquivalent.

NP-Härte von INDEPENDENT SET

Da wir nun ein NP-hartes Problem kennen, können wir aufgrund unserer vorgehenden Beobachtung NP-Härte für weitere Probleme mittels Reduktion zeigen.

INDEPENDENT SET

INSTANZ: Ungerichteter Graph $G = (V, E)$ und Integer K.

FRAGE: Existiert ein *Independent-Set* S der Größe $\geq K$ für G
d.h., gibt es eine Menge $S \subseteq V$, sodass für alle $i, j \in S$ gilt: $[i, j] \notin E$?

Theorem

3-SAT *lässt sich in polynomieller Zeit auf* INDEPENDENT SET *reduzieren.*

Folgerung:

Das INDEPENDENT SET Problem ist NP-hart. Tatsächlich ist es NP-vollständig.

Reduktion 3-SAT $\rightarrow$ INDEPENDENT SET

Beweis

Sei ϕ eine beliebige Instanz von 3-SAT.
Nehmen wir an dass ϕ m Klauseln mit jeweils exakt 3 Literalen hat.
Wir konstruieren die folgende Instanz von INDEPENDENT SET:

Ungerichteter Graph G mit einem Knoten für jedes Literal in ϕ:
$V = \{\ell_{11}, \ell_{12}, \ell_{13}, \ldots, \ell_{m1}, \ell_{m2}, \ell_{m3}\}$, also ist $|V| = 3m$.

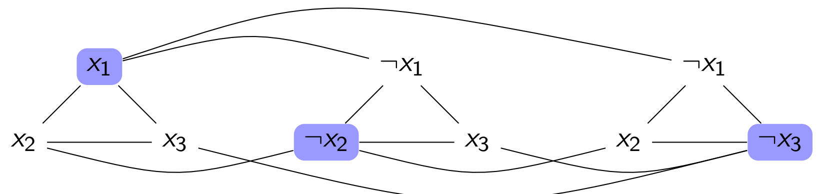
Die Knoten die Literale einer Klausel c_i von ϕ repräsentieren, sind verbunden: $E \supseteq \{[\ell_{i1}, \ell_{i2}], [\ell_{i1}, \ell_{i3}], [\ell_{i2}, \ell_{i3}]\}$ für alle i;
in anderen Worten: G hat für jede Klausel ein Dreieck.

Weiters verbinden wir alle Paare $\ell_{i\alpha}$, $\ell_{j\beta}$, wenn diese komplementäre Literale repräsentieren,
sprich: E umfasst auch alle solche Kanten $[\ell_{i\alpha}, \ell_{j\beta}]$.

Abschließend setzen wir $K = m$.

Beispiel

Die 3-KNF Formel $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$ wird auf folgenden Graph reduziert:



Der Independent-Set $S = \{\ell_{11}, \ell_{22}, \ell_{33}\}$ repräsentiert die Wahrheitswertbelegung $I(x_1) = I(\neg x_2) = I(\neg x_3) = \textbf{true}$.

Reduktion 3-SAT $\rightarrow$ INDEPENDENT SET

Beweis - Fortsetzung

Um die Korrektheit der Reduktion zu zeigen, müssen wir folgende Äquivalenz beweisen: ϕ ist erfüllbar $\Leftrightarrow$ die durch die Reduktion von ϕ gegebene Instanz (G, K) ist eine positive Instanz von $\mathrm{INDEPENDENT\ SET}$.

Wir beweisen die beiden Richtungen getrennt.

"$\Leftarrow$" Sei (G, K) eine positive Instanz von $\mathrm{INDEPENDENT\ SET}$, d.h. es gibt ein Set $S \subseteq V$, sodass für alle $i, j \in S$ $[i, j] \notin E$ gilt und $|S| \geq K = m$. Wir müssen zeigen dass ϕ eine positive Instanz von $3\text{-}\mathrm{SAT}$ ist.

Beobachtung: Die Knoten eines Dreiecks in G sind alle adjazent, daher kann S maximal einen Knoten aus jedem der m Dreiecke enthalten. Daher gilt für alle $\ell_{i\alpha}, \ell_{i\beta} \in S$, dass $\alpha = \beta$. Da $|S| \geq K$ und $K = m$, ist in S genau ein Knoten pro Dreieck. Wir identifizieren S als $\{\ell_{1\alpha_1}, \ldots, \ell_{m\alpha_m}\}$ für eine passende Kombination $\alpha_1, \ldots, \alpha_m$ für Werte aus $\{1, 2, 3\}$.

Beweis - Fortsetzung $\Leftarrow$-Richtung

Wir definieren nun eine Wahrheitswertbelegung I für ϕ:

1. Ein Atom x von ϕ ist **true** unter I wenn wir ein $\ell_{i\alpha} \in S$ haben und x ist ein positives Literal an Position α in der $i$ten Klausel von ϕ.
2. Die verbleibenden Atome werden in I auf **false** gesetzt.

Wir zeigen nun dass alle Klauseln aus ϕ zumindest ein Literal haben, welches **true** unter I ist; klarerweise ist ϕ dann erfüllbar.

Sei C eine beliebige Klausel in ϕ und nehmen wir an es ist die i-te Klausel in ϕ. Betrachte Literal $\ell_{i\alpha_i}$ in C. Wir wissen dass $\ell_{i\alpha_i} \in S$. Wir zeigen nun dass $\ell_{i\alpha_i}$ **true** in I.

Fall 1: Wenn $\ell_{i\alpha_i}$ ein positives Literal ist, d.h. ein Atom x, dann ist $\ell_{i\alpha_i}$ **true** in I da laut Definition x in I auf **true** gesetzt ist (1).

Fall 2: Sei $\ell_{i\alpha_i}$ ein negatives Literal der Form $\neg x$. Wir müssen verifizieren dass x **false** in I ist. Angenommen das ist nicht der Fall, d.h. x ist **true** in I. Dann gibt es laut Definition von I, ein $\ell_{j\beta} \in S$ wobei x ein positives Literal auf Position β in der $j$ten Klausel von ϕ ist. Daher sind $\ell_{i\alpha_i}$ und $\ell_{j\beta}$ komplementäre Literale und daher laut Reduktion $[\ell_{i\alpha_i}, \ell_{j\beta}] \in E$. Widerspruch zur Annahme dass S ein Independent-Set von G ist.

Beweis - Fortsetzung ⇒-Richtung

"⇒" Angenommen ϕ ist erfüllbar. Wir zeigen dass die reduzierte Instanz (G, K) eine positive Instanz von INDEPENDENT SET ist.

Da ϕ erfüllbar, gibt es Wahrheitswertbelegung I für ϕ, sodass $I(\phi) =$ **true**. Klarerweise macht I zumindest ein Literal in jeder Klausel von ϕ **true**. Daher können wir eine Knotenmenge $S = \{\ell_{1i_1}, \ldots, \ell_{mi_m}\}$ definieren sodass jeder Index i_j der Position eines Literals in der $j$ten Klausel von ϕ welches **true** in I ist, entspricht. (Falls mehrere Literale in einer Klausel **true** in I sind, wählen wir ein beliebiges aus). Wir müssen noch zeigen dass (i) S ein Independent-Set in G ist und (ii) $|S| \geq K$.

(ii) ist trivial: Laut Konstruktion $|S| = m$ und laut Reduktion $K = m$.

Wir zeigen (i): Angenommen dies gilt nicht, d.h. es gibt ein Paar $\ell_{i\alpha}, \ell_{j\beta} \in S$ sodass $[\ell_{i\alpha}, \ell_{j\beta}] \in E$. Laut Definition von G, hat G keine self-loops. Daher gilt $i \neq j$. Laut Reduktion stellen $\ell_{i\alpha}$ und $\ell_{j\beta}$ daher komplementäre Literale dar. Da I eine Wahrheitswertbelegung, ist eines der beiden Literale falsch unter I. Widerspruch: Laut Konstruktion haben wir für S nur Literale verwendet die **true** unter I.

Weitere NP-vollständige Graph Probleme

Die folgenden beiden Entscheidungsprobleme sind dem INDEPENDENT SET-Problem ähnlich:

CLIQUE

INSTANZ: Ungerichteter Graph $G = (V, E)$ und Integer K.

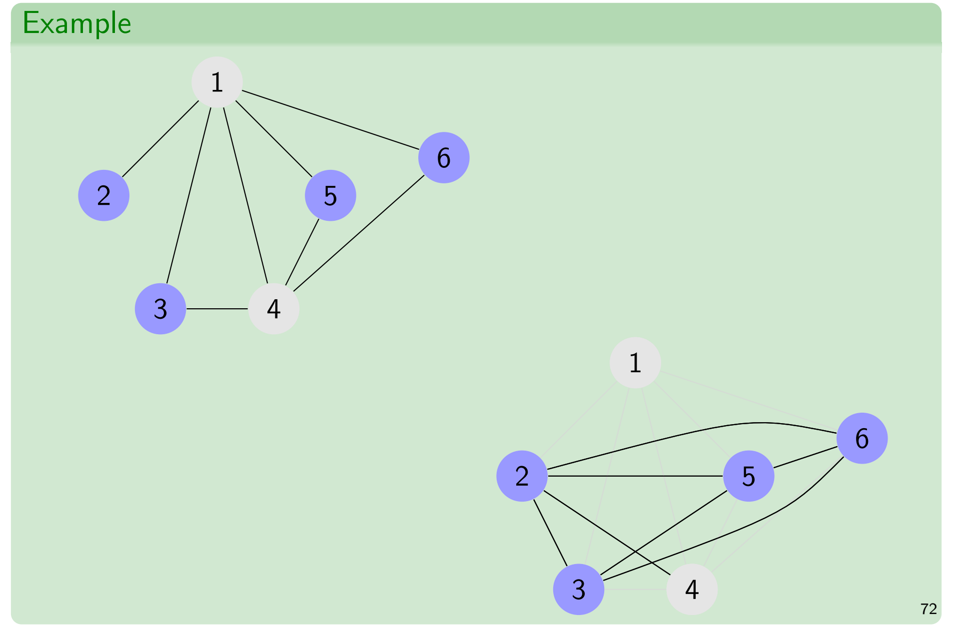
FRAGE: Enthält G eine *Clique* C der Größe $\geq K$?
D.h., gibt es ein $C \subseteq V$, sodass für alle $i, j \in C$ mit $i \neq j$, $[i, j] \in E$?

VERTEX COVER

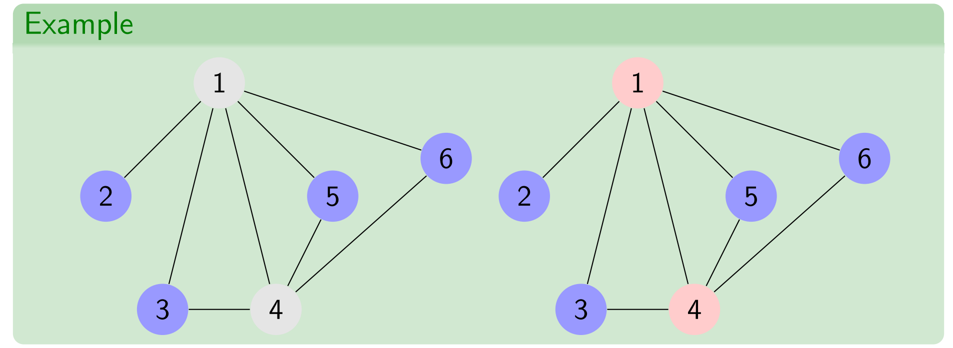
INSTANZ: Ungerichteter Graph $G = (V, E)$ und Integer K.

FRAGE: Gibt es ein *Vertex Cover* N der Größe $\leq K$ in G?
D.h., gibt es ein $N \subseteq V$, sodass für alle Kanten $[i, j] \in E$, $i \in N$ oder $j \in N$ (oder beides)?

INDEPENDENT SET vs. CLIQUE



INDEPENDENT SET vs. VERTEX COVER



Komplexität

Theorem

INDEPENDENT SET, CLIQUE, *und* VERTEX COVER *sind* NP*-vollständig.*

Proof

NP-Membership. NP-Algorithmen für diese Probleme raten zuerst eine Teilmenge S der Knoten V und checken in polynomieller Zeit dass S den jeweilegen Eigenschaften genügt. (z.B. dass S ein Independent-Set der Größe $\geq K$ ist).

NP-Härte. Die zuvor skizzierten Äquivalenzen entsprechen einfachen Reduktionen mit denen man NP-Härte für die zwei verbleibenden Probleme zeigt. (Recall: Wir haben NP-Härte für das Problem INDEPENDENT SET bereits mittels Reduktion von 3-SAT gezeigt.)

Weitere Graph Probleme

HAMILTON-PATH

INSTANZ: Graph (gerichtet oder ungerichtet) $G = (V, E)$

FRAGE: Gibt es für G einen *Hamilton'schen Pfad*, also einen Pfad der alle Knoten exakt einmal besucht?

HAMILTON-CYCLE

INSTANZ: Graph (gerichtet oder ungerichtet) $G = (V, E)$

FRAGE: Gibt es für G einen *Hamilton'schen Kreis*, also einen zylischen Pfad der alle Knoten exakt einmal besucht?

Weitere Varianten des Erfüllbarkeitsproblems

Not-all-equal SAT (NAESAT)

INSTANZ: Aussagenlogische Formel ϕ in 3-KNF

FRAGE: Gibt es eine Wahrheitswertbelegung I für ϕ, die ϕ wahr macht und in jeder Klausel max. zwei der drei Literale auf wahr setzt?

1-IN-3-SAT

INSTANZ: Aussagenlogische Formel ϕ in 3-KNF

FRAGE: Gibt es eine Wahrheitswertbelegung I für ϕ, die in jeder Klausel genau eines der 3 Literale auf wahr setzt?

NP-Vollständige Probleme

Theorem

Die folgenden Probleme sind alle NP*-vollständig.*

- K-COL *für jedes* $k \geq 3$ *(also insbesonders auch* 3-COL*)*
- HAMILTON-PATH, HAMILTON-CYCLE
- K-SAT *für jedes* $k \geq 3$, NAESAT, 1-IN-3-SAT

Noch einmal: Varianten des Erfüllbarkeitsproblems

Anmerkungen

- Für positve Instanzen gilt $\mathrm{1\text{-}IN\text{-}3\text{-}SAT} \subset \mathrm{NAESAT} \subset \mathrm{3\text{-}SAT}$. Die Menge aller Instanzen dieser 3 Probleme ist aber gleich. In anderen Worten, es ist die Frage die sich hier ändert.
- Daher folgt aus der NP-Vollständigkeit eines dieser 3 Probleme nicht die NP-Vollständigkeit eines der anderen Probleme (a priori ist es nicht klar ob das Beschränken der positiven Instanzen das Problem leichter oder schwieriger macht!)
- Im Gegensatz dazu ist $\mathrm{3\text{-}SAT}$ ein Spezialfall von SAT, d.h., die Instanzen des einen Problems sind sind eine Teilmenge der Instanzen des anderen Problems — bei gleicher Fragestellung.
  1. NP-Härte des Spezialfalls impliziert NP-Härte des allgemeineren Problems
  2. NP-Membership des allgemeineren Problems impliziert NP-Membership des Spezialfalls

NP-Vollständigkeit – Spezialfall

Wir nennen einen ungerichteten Graphen G non-terminal wenn jeder Knoten in G zumindest zwei Kanten zu anderen Knoten hat. Beispiele: $(\{a, b, c\}, \{[a, b], [b, c], [a, c]\})$ ist non-terminal, während $(\{a, b, c\}, \{[a, b], [b, c]\})$ oder $(\{a, b, c, d\}, \{[a, b], [b, c], [a, c]\})$ diese Eigenschaft nicht haben.

Betrachten wir das folgende Entscheidungsproblem:

3-COL-NT

INSTANZ: Ein non-terminal Graph $G = (V, E)$.

FRAGE: Ist der Graph G 3-färbbar?
d.h.: gibt es eine Funktion $f : V \to \{0, 1, 2\}$, so dass für alle Kanten $[v_i, v_j] \in E$ gilt: $f(v_i) \neq f(v_j)$.

Wir nutzen die Tatsache dass die Standard Version des 3-Färbbarkeitsproblems NP-vollständig ist.

NP-Vollständigkeit – Spezialfall

Für die NP-Vollständigkeit von $3\text{-}\mathrm{COL}\text{-}\mathrm{NT}$ müssen wir NP-membership und NP-Härte zeigen.

NP-membership: $3\text{-}\mathrm{COL}\text{-}\mathrm{NT}$ ist ein Spezialfall von $3\text{-}\mathrm{COL}$, d.h. jede Instanz von $3\text{-}\mathrm{COL}\text{-}\mathrm{NT}$ ist auch Instanz von $3\text{-}\mathrm{COL}$. Da $3\text{-}\mathrm{COL} \in$ NP, folgt $3\text{-}\mathrm{COL}\text{-}\mathrm{NT} \in$ NP direkt.

NP-Vollständigkeit – Spezialfall

NP-Härte: Wir geben eine polynomielle many-one Reduktion von 3-COL auf 3-COL-NT.
Sei $G = (V, E)$ eine beliebige Instanz von 3-COL, also ein beliebiger ungerichteter Graph. Wir konstruieren eine Instanz $G' = (V', E')$ von 3-COL-NT mit

- $V' = \{v, v^1, v^2 \mid v \in V\}$ und
- $E' = E \cup \{[v, v^1], [v^1, v^2], [v, v^2] \; midv \in V\}$.

Laut Definition ist jeder solcher Graph G' non-terminal und wir haben somit eine Reduktion wie gefordert. Außerdem ist klar, dass die Reduktion in polynomieller Zeit berechnet werden kann.

Wir müssen noch die Korrektheit dieser Reduktion zeigen, also: G ist positive Instanz von $3\text{-COL} \Leftrightarrow G'$ ist positive Instanz von 3-COL-NT.

NP-Vollständigkeit – Spezialfall

$\Rightarrow$: Sei $G = (V, E)$ eine positive Instanz von 3-COL. Dann gibt es eine Funktion f die den Knoten aus V einen Wert aus $\{0, 1, 2\}$ so zuweist, dass $f(u) \neq f(v)$ für alle Kanten $[u, v] \in E$ gilt.

Wir konstruieren nun eine Färbung $f^* : V' \rightarrow \{0, 1, 2\}$ für G' wie folgt:

$$\text{für alle } v \in V, \text{ sei } f^*(v) = f(v) \text{ und } f^*(v^i) = (f(v) + i) \textit{ mod } 3.$$

Wir müssen zeigen dass für allen Kanten $[x, y] \in E'$, $f^*(x) \neq f^*(y)$. Wir unterscheiden dabei die folgenden Fälle:

(1) Betrachte zuerst $x, y \in V$: Laut Annahme ist f eine 3-Färbung für G; laut Definition von f^* gilt daher $f^*(x) \neq f^*(y)$.

(2) Laut Konstruktion gilt andererseits $x, y \in \{v, v^1, v^2\}$ für ein $v \in V$; laut Definition von f^* haben wir unmittelbar $f^*(x) \neq f^*(y)$. Es folgt dass G' 3-färbbar ist und daher eine positive Instanz von 3-COL-NT.

NP-Vollständigkeit – Spezialfall

$\Leftarrow$: Sei G' eine positive Instanz von 3-COL-NT. Da G' 3-färbbar ist auch jeder Subgraph von G' 3-färbbar. Da G ein solcher Subgraph ist, gilt unmittelbar, dass G' eine positive Instanz von 3-COL.

Wir haben somit die Korrektheit unserer Reduktion gezeigt.

Teil 3: Einführung in Komplexitätstheorie

Teil 3.1: Grundlegende Konzepte und Wiederholung

Teil 3.2: Die Komplexitätsklassen P und NP

Teil 3.3: NP-Vollständigkeit

Teil 3.4: Weitere Komplexitätsklassen

The Klasse L (1)

Definition von L

L ist die Klasse jener Probleme die mittels eines SIMPLE Programms gelöst werden können welches lediglich logarithmischen Speicherbedarf hat, d.h. $\mathcal{P} \in \mathrm{L}$ wenn es ein Programm gibt sodass für alle Instanzen I von $\mathcal{P}$ dieses Programm $\mathrm{O}(log_2|I|)$ Bits im Arbeitsspeicher braucht.

- Gemäß Definition von $\mathrm{O}(log_2|I|)$ heisst das, dass das Programm max. $c \cdot log_2|I| + d$ Bits des Arbeitspeicher braucht, wobei c, d Konstante.
- Logarithmic space = really little space, z.B. $log_2(65536) = 16$.
- Wir dürfen also im Programm nur konstant viele Pointer (Addressen im Speicher), Zähler, und logarithmisch viele Boole'sche Variable nutzen.
- Die Laufzeit ist apriori nicht eingeschränkt!
  (aber ist maximal polynomiell, wie wir sehen werden).

Die Klasse L (2)

- Für ein logarithmic-space Programm nehmen wir an dass der Input in einem separaten Read-only Memory abgelegt wird.
- Wir messen lediglich die Bits im Arbeitsspeicher
  - Klarerweise gilt $c \cdot log_2 n + d < n$ für ein ausreichend großes n.
  - Daher brauchen wir diesen separaten Speicher für den Input, da wir unter den gegebenen Speicherbeschränkungen sonst nicht einmal den Input parsen könnten.

- Ein Beispielproblem für L:

FIND-NODE

INSTANZ: Natürliche Zahl n und Baum T, wobei jeder Baumknoten mit einer natürlichen Zahl gelabelt ist.

FRAGE: Enthält T einen Knoten mit Label n?

- **FIND-NODE** $\in$ L da wir T depth-first mit nur drei Pointer durchlaufen können: *current*, *next*, und *aux*.

FIND-NODE in Logarithmic Space

Wir können die folgenden Subroutinen voraussetzen (laufen in log. space):

- $root(T)$ zeigt auf Wurzel von T
  - $root(T) = nil$ wenn T der leere Baum
- $firstChild(T, x)$ liefert erstes Kind des Knoten x in T
  - $firstChild(T, x) = nil$ wenn ein solches Kind nicht existiert
- $rightSibling(T, x)$ liefert den rechten Nachbarn von x in T
  - $rightSibling(T, x) = nil$ wenn dieser nicht existiert
- $parent(T, x)$ liefert den Elternknoten von x in T
  - $parent(T, x) = nil$ wenn x Wurzel von T
- $isChildOf(T, x_1, x_2)$ liefert **true** wenn x_1 Kind von x_2 in T
- $isLeaf(T, x)$ liefert **true** wenn $firstChild(T, x) = nil$
- $hasRightSibling(T, x)$ liefert **true** wenn $rightSibling(T, x) \neq nil$
- $labelling(T, x)$ liefert das Label von x in T

// Speichereffiziente Prozedur eines depth-first Durchlaufs

```
Boolean find(Tree T, Integer val)
  current = root(T);
  if labelling(T, current) = val then return true;
  next = firstChild(T, current);

  while (next != nil) {

    if isChildOf(T, next, current) and !isLeaf(T, next) then {
      current := next; next := firstChild(T, next) }

    else if isChildOf(T, next, current) and isLeaf(T, next) then {
      aux := current; current := next; next := aux }

    else if isChildOf(T, current, next) and hasRightSibling(T, current) then
      { aux := current; current := next; next := rightSibling(T, aux) }

    else if isChildOf(T, current, next) and !hasRightSibling(T, current) {
      current := next; next := parent(T, next) }

    if labelling(T, current) = val then return true
  }
  return false
```

L und P

Theorem

L $\subseteq$ P, *also jedes Problem mit logarithmischem Specherbedarf ist in Polynomialzeit lösbar.*

Beweisidee

Zur Erinnerung: ein Programm Π dass logarithmischen Speicherbedarf in Bezug auf Input I hat, verbraucht maximal $c \cdot log_2|I| + d$ Bits im Read/Write Memory, mit c, d Konstante. Jeder Berechnungszustand (Konfiguration) des Programms im Zuge der Ausführung von Π auf I muss eindeutig durch die Belegung des Speichers beschreibbar sein. Wir haben daher dass es max. $2^{c \cdot log_2|I|+d}$ mögliche Zustände gibt, in denen sich Π befinden kann. Da $2^{c \cdot log_2|I|+d} = (2^{\log_2 |I|})^c \cdot 2^d = |I|^c \cdot 2^d$, folgt dass es maximale polynomiell viele Zustände (in Bezug auf $|I|$) für Π geben kann. Folgerichtig kann die Laufzeit von Π auch nur polynomiell sein.

Die Klasse PSPACE

PSPACE

PSPACE ist die Klasse jener Probleme die mittels eines SIMPLE Programms gelöst werden können welches polynomielen Speicherbedarf hat, d.h. $\mathcal{P} \in$ PSPACE wenn es ein Programm gibt sodass für alle Instanzen I von $\mathcal{P}$ dieses Programm $\mathrm{O}(|I|^k)$ Bits im Arbeitsspeicher braucht (k eine Konstante).

- Diese Klasse ist sehr umfangreich und enthält zahlreiche schwierige Probleme.
- "Polynomieller Speicherbedarf" bedeutet dass das Programm in exponentiell vielen unterschiedlichen Zuständen sein kann (das Arugment ist analog zu jenem bzgl. den polynomiell vielen Zuständen im Fall von L)
- PSPACE-vollständige Probleme gelten als intractable (wir werden noch sehen dass NP $\subseteq$ PSPACE).

Beispiel eines Problems in PSPACE: Tic-Tac-Toe

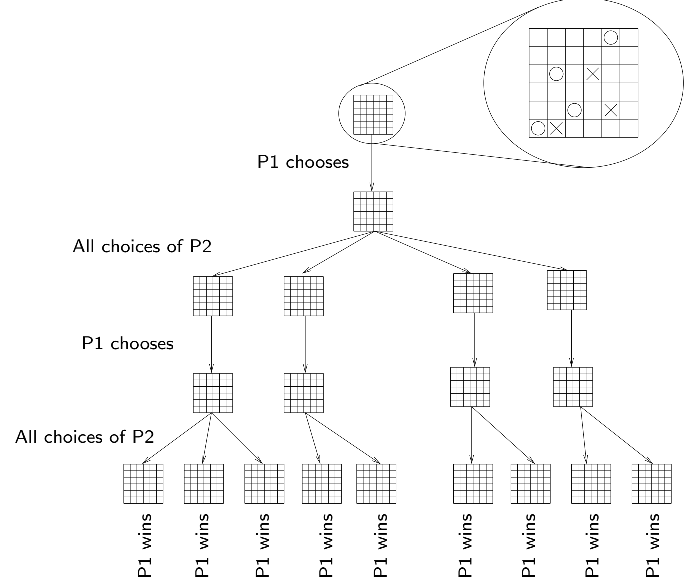
- Wir verallgemeinern das klassische (3×3 Spielbrett) Tic-Tac-Toe Spiel
- Ein (n, k)-Tic-Tac-Toe bezeichnet Tic-Tac-Toe
  - auf einem $n \times n$ Spielbrett;
  - man gewinnt wenn man eine Reihe der Länge k legt.
- Beobachtung: ein Spiel dauert max. n^2 steps.

TTT-WINNING-STRATEGY

INSTANZ: Zahlen n, k, ein Spielstand C auf dem $n \times n$ Brett.

FRAGE: Hat Spieler 1 eine Gewinnstrategie für das (n, k)-Tic-Tac-Toe ausgehend von Spielstand C?

Gewinnstrategie für Verallgemeinertes Tic-Tac-Toe



TTT-WINNING-STRATEGY ist PSPACE-vollständig

TTT-WINNING-STRATEGY

Die Problem ist in PSPACE (tatsächlich ist es PSPACE-vollständig):

- er reicht im Speicher einen Stack von max. n^2 Konfigurationen zu halten;
- jede Konfiguration braucht nur polynomiellen Speicher.

Anmerkungen

- Alternation ist typisch für PSPACE-vollständige Probleme, also:
  es gibt einen Zug für Spieler P1, sodass
  für alle möglichen Züge von P2,
  es gibt einen Zug für P1, ...
- Probleme in PSPACE können mittels polynomiell tief geschachtelten Schleifen gelöst werden: Im Fall von TTT-WINNING-STRATEGY ist die Tiefe mit n^2 beschränkt.

Weitere Probleme in PSPACE

SQL Query

Auswerten einer SQL Query über einer Datenbank.

Kontextsensitive Grammatik

Entscheiden ob ein gegebenes Wort von einer gegebenen kontextsensitiven Grammatik erzeugt werden kann.

Universalität eines regulären Ausdrucks

Testen ob ein regulärer Ausdruck alle möglichen Strings erzeugt.

PSPACE vs. P

Theorem

P $\subseteq$ PSPACE: *jedes Problem dass sich in polynomieller Zeit lösen lässt, braucht max. polynomiellen Speicher.*

Beweis

Offensichtlich: In polynomieller Zeit ist es nicht möglich exponentiell viel Speicher zu nutzen/allokieren.

PSPACE vs. NP

Theorem

NP $\subseteq$ PSPACE.

Beweis Idee

Sei $\mathcal{P} \in$ NP. Dann gibt es eine Zertifikatsrelation R für $\mathcal{P}$, die polynomiell balanziert und polynomiel entscheidbar ist.

Wir können relativ einfach ein Programm spezifieren, welches Instanzen I von $\mathcal{P}$ in polynomiellen Speicherbedarf löst.

- Gehe alle Zertifikat-Kanditaten einzeln durch (klarerweise ist dann nur jeweils ein Kandidat im Speicher zu halten). Da R polynomiell balanziert, wissen wir dass ein solcher Kandidat nur polynomiellen Speicher braucht.
- Für jeden Kandidat C, teste $(I, C) \in R$. Da R polynomiell entscheidbar, braucht ein solcher Test nur polynomielle Zeit und daher, wie wir bereits gesehen haben, geht auch das mit polynomiellen Speicherbedarf.

Die Klasse EXPTIME

EXPTIME

Die Klasse EXPTIME beinhaltet genau jene Entscheidungsprobleme $\mathcal{P}$ die folgende Eigenschaften haben:

1. es gibt ein SIMPLE Programm Π für $\mathcal{P}$, sodass
2. für alle Instanzen I von $\mathcal{P}$ die Laufzeit von Π auf I exponentiell in $|I|$ ist, d.h. die Laufzeit ist durch $\mathrm{O}(2^{|I|^k})$, wobei k eine Konstante, beschränkt.

Die folgenden Beziehungen sind bekannt:

- PSPACE $\subseteq$ EXPTIME (analog zu L $\subseteq$ P)
- P $\subsetneq$ EXPTIME (nicht-triviales Diagonalisierungs-Argument)

Beispiele für Probleme in EXPTIME:

- Existenz einer Gewinnstrategie in GO auf einem $n \times n$ Spielbrett (Begründung: es existiert keine polynomielle Schranke für die Dauer des Spiels),
- Auswertung von DATALOG queries (SQL mit Rekursion).

Diskussion

Es gibt noch zahlreiche weitere Klassen:

- 2-EXPTIME $=$ Probleme die in Zeit $\mathrm{O}(2^{2^{|I|^k}})$ lösbar
- 3-EXPTIME $=$ Probleme die in Zeit $\mathrm{O}(2^{2^{2^{|I|^k}}})$ lösbar
- ...
- EXPSPACE $=$ Probleme die mit Speicher $\mathrm{O}(2^{|I|^k})$ lösbar
- 2-EXPSPACE $=$ Probleme die mit Speicher $\mathrm{O}(2^{2^{|I|^k}})$ lösbar
- 3-EXPSPACE $=$ Probleme die mit Speicher $\mathrm{O}(2^{2^{2^{|I|^k}}})$ lösbar
- ...

Nichtdeterminismus und Platzbedarf

Wir können analog zu NP auch Klassen wie z.B. NL, NPSPACE oder NEXPSPACE definieren.

Klarerweise gilt:

$$\mathsf{L} \subseteq \mathsf{NL} \quad \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \quad \mathsf{EXPSPACE} \subseteq \mathsf{NEXPSPACE}$$

Es ist offen ob $\mathsf{L} = \mathsf{NL}$ gilt (vgl. P vs. NP)

Man kann aber zeigen:

$$\mathsf{PSPACE} = \mathsf{NPSPACE} \quad \mathsf{EXPSPACE} = \mathsf{NEXPSPACE}$$

Beziehung zwischen Komplexitätsklassen

Theorem

$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME} \subseteq \mathsf{NEXPTIME} \subseteq \mathsf{EXPSPACE}$

Bzgl. echter Teilmengeninklusion ist nicht mehr bekannt als:
$\mathsf{L} \subset \mathsf{PSPACE}$, $\mathsf{P} \subset \mathsf{EXPTIME}$, $\mathsf{PSPACE} \subset \mathsf{EXPSPACE}$.

Komplementäre Probleme und Komplexitätsklassen

Gegeben ein Entscheidungsproblem $\mathcal{P}$ mit Instanzen I und positive Instanzen $Y \subseteq I$. Das Komplementärproblem $\overline{\mathcal{P}}$ ist definiert über Instanzen I und positive Instanzen $I \setminus Y$ (informell: die Frage des Entscheidungsproblem $\mathcal{P}$ wird verneint).

Definition

Jede Komplexitätsklasse $\mathcal{C}$ hat eine Komplementärklasse, co-$\mathcal{C}$ die wie folgt definiert ist:

- co-$\mathcal{C} = \{\overline{\mathcal{P}} \mid P \in \mathcal{C}\}$.

Anmerkungen

- Beispiele: co-P, co-NP, co-NEXPTIME, etc.
- Deterministische Komplexitätsklassen $\mathcal{C}$ sind abgeschlossen unter Komplement, d.h. sie stimmen mit der co-Klasse co-$\mathcal{C}$ überein, z.B: P = co-P.
- Für nichtdeterministische Komplexitätsklassen $\mathcal{C}$ gilt das i.A. nicht. So ist z.B. offen ob NP = co-NP oder NP $\neq$ co-NP.

Zusammenfassung von Teil 3 der LVA

- Wiederholung grundlegender Konzepte, insbes. Reduktionen
- Korrektheitsbeweis von Reduktionen
- zentrale Komplexitätsklassen: P und NP
- Verschiedene Charakterisierungen von NP
- NP-Vollständigkeit
- SAT als prototypisches NP-vollständiges Problem
- Beispiele fuer NP-vollständige Probleme
- weitere Komplexitätsklassen

Zusammenfassung von Teil 3 der LVA

... als Video:

https://www.youtube.com/watch?v=YX40hbAHx3s

6.0 VU Theoretische Informatik (192.017)

Teil 4: Formale Semantik von Programmen

Gernot Salzer

Institut für Logic and Computation

Wintersemester 2023

Ist dieses Programm korrekt?

```
// Multipliziert x und y
z := 0;
while y ≠ 0 do {
   z := z + x;
   y := y − 1
}
```

Schwer zu sagen:

- Ist das ein zulässiges Programm? Was alles ist zulässig? ⇒ Syntax
- Was bedeuten die Programmelemente? ⇒ operationale Semantik
- Wie ist die Behauptung „Multipliziert x und y" zu interpretieren? ⇒ Korrektheitsaussagen
- Wie können wir so eine Behauptung (automatisch) beweisen? ⇒ axiomatische Semantik (Hoare-Kalkül)

SIMPLE($\mathbb{Z}$): SIMPLE über den ganzen Zahlen – Syntax

Spezifikationsmethode: kontextfreie Produktionen

Programme

Programm $\rightarrow$	ε	Leerprogramm
	| "`abort`"	Abbruch, Exception
	| *Var* ":=" *Term*	Zuweisung
	| "`{`" *Programm* "`;`" *Programm* "`}`"	Hintereinanderausführung
	| "`if`" *Term* "`then`" *Programm* "`else`" *Programm*	Konditional
	| "`while`" *Term* "`do`" *Programm*	Schleife

Variablen

$Var \rightarrow$ "x" | "y" | "z" | "x_1" | $\cdots$ | "*alles_was_gut_und_teuer_ist*" | $\cdots$

$\mathcal{L}(Programm)$ ... Menge aller Programme

$\mathcal{L}(Var) = \{x, y, z, x_1, \ldots\}$ ... Menge aller Variablen

SIMPLE($\mathbb{Z}$) – Syntax (2)

Terme

$$
\begin{aligned}
\mathit{Term} &\rightarrow \mathit{Var} \mid \mathit{Const} \mid \mathit{UnOp}\ \mathit{Term} \mid \text{“(”}\ \mathit{Term}\ \mathit{BinOp}\ \mathit{Term}\ \text{“)”} \\
\mathit{UnOp} &\rightarrow \text{“+”} \mid \text{“−”} \mid \text{“}\neg\text{”} \\
\mathit{BinOp} &\rightarrow \text{“+”} \mid \text{“−”} \mid \text{“}*\text{”} \mid \text{“/”} \mid \text{“}\wedge\text{”} \mid \text{“}\vee\text{”} \mid \text{“}\supset\text{”} \\
&\quad \mid \text{“}=\text{”} \mid \text{“}\neq\text{”} \mid \text{“}>\text{”} \mid \text{“}\geq\text{”} \mid \text{“}<\text{”} \mid \text{“}\leq\text{”} \\
\mathit{Const} &\rightarrow \text{“0”} \mid \text{“1”} \mid \text{“2”} \mid \cdots \mid \text{“42”} \mid \cdots
\end{aligned}
$$

Konventionen:

- Kommentare beginnen mit //.
- Whitespace (Leerzeichen, Zeilenumbrüche, Kommentare) beliebig.
- Manche Klammern können fehlen: Außenklammern, assoziative Operatoren, Punkt- vor Strichrechnung

Beispiel: Multiplikationsprogramm in SIMPLE($\mathbb{Z}$)

$$\begin{aligned}
\mathit{Programm} &\Rightarrow_P \text{“\{”}\ \mathit{Programm}\ \text{“;”}\ \mathit{Programm}\ \text{“\}”} \\
&\Rightarrow_P \text{“\{”}\ \mathit{Var}\ \text{“:=”}\ \mathit{Term}\ \text{“;} \\
&\qquad \texttt{while}\text{”}\ \mathit{Term}\ \text{“}\texttt{do}\text{”}\ \mathit{Programm}\ \text{“\}”} \\
&\Rightarrow_P \text{“\{ } z \coloneqq \text{”}\ \mathit{Const}\ \text{“;} \\
&\qquad \texttt{while (}\text{”}\ \mathit{Term}\ \mathit{BinOp}\ \mathit{Term}\ \text{“)}\ \texttt{do \{}\text{”} \\
&\qquad\quad \mathit{Programm}\ \text{“;”} \\
&\qquad\quad \mathit{Programm}\ \text{“\} \}”} \\
&\Rightarrow_P \text{“\{ } z \coloneqq 0; \\
&\qquad \texttt{while (}\text{”}\ \mathit{Var}\ \text{“}\neq\text{”}\ \mathit{Const}\ \text{“)}\ \texttt{do \{}\text{”} \\
&\qquad\quad \mathit{Var}\ \text{“:=”}\ \mathit{Term}\ \text{“;”} \\
&\qquad\quad \mathit{Var}\ \text{“:=”}\ \mathit{Term}\ \text{“\} \}”} \\
&\Rightarrow_P \text{“\{ } z \coloneqq 0; \\
&\qquad \texttt{while}\ (y \neq 0)\ \texttt{do \{} \\
&\qquad\quad z \coloneqq \text{(”}\ \mathit{Term}\ \mathit{BinOp}\ \mathit{Term}\ \text{“)}\,; \\
&\qquad\quad y \coloneqq \text{(”}\ \mathit{Term}\ \mathit{BinOp}\ \mathit{Term}\ \text{“)\} \}”}
\end{aligned}$$

Beispiel: Multiplikationsprogramm in SIMPLE($\mathbb{Z}$) (2)

$$
\begin{aligned}
\textit{Programm} &\Rightarrow_P \cdots \\
&\Rightarrow_P \text{“}\{\ z := 0; \\
&\qquad \texttt{while}\ (y \neq 0)\ \texttt{do}\ \{ \\
&\qquad\quad z := (\text{”}\ \textit{Term BinOp Term}\ \text{“})\,; \\
&\qquad\quad y := (\text{”}\ \textit{Term BinOp Term}\ \text{“})\}\ \}\text{”} \\
&\Rightarrow_P \text{“}\{\ z := 0; \\
&\qquad \texttt{while}\ (y \neq 0)\ \texttt{do}\ \{ \\
&\qquad\quad z := (\text{”}\ \textit{Var}\ \text{“}+\text{”}\ \textit{Var}\ \text{“})\,; \\
&\qquad\quad y := (\text{”}\ \textit{Var}\ \text{“}-\text{”}\ \textit{Const}\ \text{“})\}\ \}\text{”} \\
&\Rightarrow_P \text{“}\{\ z := 0; \\
&\qquad \texttt{while}\ (y \neq 0)\ \texttt{do}\ \{ \\
&\qquad\quad z := (z + x)\,; \\
&\qquad\quad y := (y - 1)\}\ \}\text{”}
\end{aligned}
$$

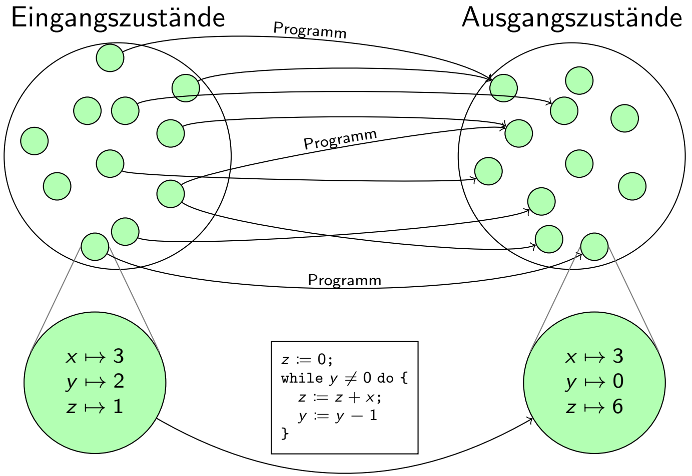
```
z := 0;
while y ≠ 0 do {
  z := z + x;
  y := y − 1
}
```

Ist dieses Programm korrekt?

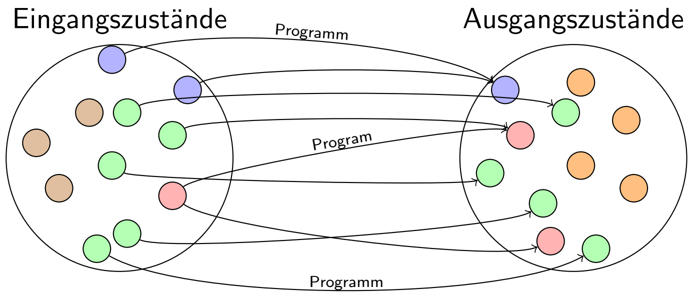
```
// Multipliziert x und y
z := 0;
while y ≠ 0 do {
  z := z + x;
  y := y − 1
}
```

- ✓ Ist das ein zulässiges Programm? Was alles ist zulässig? ⇒ Syntax
- Was bedeuten die Programmelemente? ⇒ operationale Semantik
- Wie ist die Behauptung „Multipliziert x und y“ zu interpretieren? ⇒ Korrektheitsaussagen
- Wie können wir so eine Behauptung (automatisch) beweisen? ⇒ axiomatische Semantik (Hoare-Kalkül)

Programme als Zustandstransformatoren



Programme als Zustandstransformatoren



- Mehrere Eingangszustände können in denselben Ausgangszustand führen.
- Manche Eingangszustände führen zu keinem Ausgangszustand. (Programm bricht ab oder loopt.)
- Manche Ausgangszustände sind von keinem Eingangszustand erreichbar.
- Ein Eingangszustand kann zu mehreren Ausgangszuständen führen (Indeterminismus).

Programmzustand

Informell: Momentaufnahme des Speichers (Snapshot)

Formal: Abbildung von Variablen auf Werte (hier: ganze Zahlen)

Menge aller Zustände: $\mathcal{S} \stackrel{\text{def}}{=} \{\sigma \mid \sigma\colon \mathcal{L}(\mathit{Var}) \mapsto \mathbb{Z}\}$

Unser Beispielprogramm Π bildet den Zustand σ auf den Zustand σ' ab:

$$\begin{array}{ccc} \sigma(x) = 3 & & \sigma'(x) = 3 \\ \sigma(y) = 2 & \stackrel{\Pi}{\longrightarrow} & \sigma'(y) = 0 \\ \sigma(z) = 1 & & \sigma'(z) = 6 \end{array}$$

Wir schreiben das als $[\Pi](\sigma) = \sigma'$ oder $[\Pi]\,\sigma = \sigma'$.

$[\Pi]\colon \mathcal{S} \mapsto \mathcal{S}$ ... (partielle) Funktion berechnet durch das Programm Π
Semantik (= Bedeutung) von Π

Konfiguration

Informell: Momentaufnahme des Systems („Dump“); bestehend aus

- dem verbleibenden Programm, das noch ausgeführt werden muss und
- der Momentaufnahme des Speichers (Zustand).

Formal:

- ein Paar (Π, σ), wobei $\Pi \in \mathcal{L}(\mathit{Programm})$ und $\sigma \in \mathcal{S}$,
- oder nur ein Zustand σ (finale Konfiguration).

Menge der Konfigurationen:

$$\mathcal{C} \stackrel{\text{def}}{=} (\mathcal{L}(\mathit{Programm}) \times \mathcal{S}) \cup \mathcal{S}$$

Übergänge

Übergangsrelation $\Rightarrow$: beschreibt einen einzelnen Berechnungsschritt, führt eine nicht-finale Konfiguration in die Nachfolgekonfiguration über.

$$\Rightarrow \quad \subseteq \quad (\mathcal{L}(\mathit{Programm}) \times \mathcal{S}) \times \mathcal{C}$$

Typische Situationen:

- $(\Pi, \sigma) \Rightarrow (\Pi', \sigma')$: Berechnungsschritt mittendrinnen
- $(\Pi, \sigma) \Rightarrow \sigma'$: letzter Berechnungsschritt

Eine Übergangsrelation heißt

- deterministisch, wenn es für jede Konfiguration höchstens eine Nachfolgekonfiguration gibt, d.h.,
  wenn es zu jedem $\gamma \in \mathcal{C}$ höchstens ein γ' gibt, sodass $\gamma \Rightarrow \gamma'$.
- indeterministisch, wenn es mindestens eine Konfiguration mit unterschiedlichen Nachfolgekonfigurationen gibt, d.h.,
  wenn es Konfigurationen $\gamma, \gamma', \gamma'' \in \mathcal{C}$ gibt, sodass $\gamma \Rightarrow \gamma'$, $\gamma \Rightarrow \gamma''$ und $\gamma' \neq \gamma''$ gilt.

Übergangsrelation für SIMPLE($\mathbb{Z}$) (1)

ε (Leerprogramm): lässt den Zustand unverändert

$$(\varepsilon, \sigma) \Rightarrow \sigma$$

`abort`: bricht die Berechnung ab, keine Folgekonfiguration definiert.

$$(\texttt{abort}, \sigma) \not\Rightarrow \ldots$$

(Minimales Exception Handling)

`{ Π ; Ω }`: Ausführung von links nach rechts, beginnend mit der ersten Anweisung von Π

$$\frac{(\Pi, \sigma) \Rightarrow (\Pi', \sigma')}{(\{\,\Pi\,;\Omega\,\}, \sigma) \Rightarrow (\{\,\Pi'\,;\Omega\,\}, \sigma')} \qquad \frac{(\Pi, \sigma) \Rightarrow \sigma'}{(\{\,\Pi\,;\Omega\,\}, \sigma) \Rightarrow (\Omega, \sigma')}$$

Lies: Wenn (Π, σ) in einem Schritt nach (Π', σ') führt,
dann führt $(\{\,\Pi\,;\Omega\,\}, \sigma)$ in einem Schritt nach $(\{\,\Pi'\,;\Omega\,\}, \sigma')$.

Beispiel: Übergangsrelation

Sei σ irgendein Zustand.

$$\frac{(\varepsilon, \sigma) \Rightarrow \sigma}{(\{\varepsilon\texttt{;}\{\varepsilon\texttt{;abort}\}\}, \sigma) \Rightarrow (\{\varepsilon\texttt{;abort}\}, \sigma)}$$

$$\frac{\dfrac{(\varepsilon, \sigma) \Rightarrow \sigma}{(\{\varepsilon\texttt{;}\varepsilon\}, \sigma) \Rightarrow (\varepsilon, \sigma)}}{(\{\{\varepsilon\texttt{;}\varepsilon\}\texttt{;abort}\}, \sigma) \Rightarrow (\{\varepsilon\texttt{;abort}\}, \sigma)}$$

$$\frac{\dfrac{(\texttt{abort}, \sigma) \not\Rightarrow \ldots}{(\{\texttt{abort;}\varepsilon\}, \sigma) \not\Rightarrow \ldots}}{(\{\{\texttt{abort;}\varepsilon\}\texttt{;}\varepsilon\}, \sigma) \not\Rightarrow \ldots}$$

Kein Rechenschritt möglich.

Übergangsrelation für SIMPLE($\mathbb{Z}$) (2)

$\texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}\ \Omega$: führe Π aus, wenn der Term e zu wahr evaluiert, andernfalls führe Ω aus.

$$\frac{[e]\,\sigma \neq 0}{(\texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}\ \Omega, \sigma) \Rightarrow (\Pi, \sigma)} \qquad \frac{[e]\,\sigma = 0}{(\texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}\ \Omega, \sigma) \Rightarrow (\Omega, \sigma)}$$

- $[e] : \mathcal{S} \mapsto \mathbb{Z}$ bezeichnet die durch den Term e berechnete Funktion. Bildet den momentanen Zustand auf den Wert des Terms im momentanen Zustand ab. (Siehe unten.)
- Design Entscheidung: 0 repräsentiert falsch, jeder andere Wert (insbesondere 1) repräsentiert wahr.
  Vorteil: Wir brauchen keinen zweiten Datentyp „boolean“.
- If-then ohne else:
  $\texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}\ \varepsilon = \texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}$

Beispiel: Übergangsrelation

Sei $\sigma(x) = 3$.

$$\frac{[x = 3]\sigma = 1 \neq 0}{(\texttt{if } x = 3 \texttt{ then abort else } \varepsilon, \sigma) \Rightarrow (\texttt{abort}, \sigma)}$$

Sei $\sigma(x) = 5$.

$$\frac{[x = 3]\sigma = 0}{(\texttt{if } x = 3 \texttt{ then abort else } \varepsilon, \sigma) \Rightarrow (\varepsilon, \sigma)}$$

Übergangsrelation für SIMPLE($\mathbb{Z}$) (3)

`while` e `do` Π:

Tue nichts, wenn e zu falsch evaluiert.

$$\frac{[e]\,\sigma = 0}{(\texttt{while}\ e\ \texttt{do}\ \Pi, \sigma) \Rightarrow \sigma}$$

Andernfalls führe Π aus und wiederhole die `while`-Anweisung.

$$\frac{[e]\,\sigma \neq 0}{(\texttt{while}\ e\ \texttt{do}\ \Pi, \sigma) \Rightarrow (\texttt{\{}\,\Pi\,\texttt{;while}\ e\ \texttt{do}\ \Pi\,\texttt{\}}, \sigma)}$$

Beispiel: Übergangsrelation

Sei $\tau(x) = 0$.

$$\frac{[x > 0]\tau = 0}{(\texttt{while } x > 0 \texttt{ do } x := x - 1, \tau) \Rightarrow \tau}$$

Sei $\tau(x) = 1$.

$$\frac{[x > 0]\tau = 1 \neq 0}{\begin{array}{l}(\texttt{while } x > 0 \texttt{ do } x := x - 1, \tau) \\ \Rightarrow (\{x := x - 1; \texttt{while } x > 0 \texttt{ do } x := x - 1\}, \tau)\end{array}}$$

Übergangsrelation für SIMPLE($\mathbb{Z}$) (4)

$v := e$: Der neue Zustand sieht so aus wie der alte, außer dass die Variable v den Wert des Terms e (im alten Zustand) besitzt.

$$(v := e, \sigma) \Rightarrow \sigma' \qquad \text{wobei } \sigma'(w) = \begin{cases} [e]\,\sigma & \text{wenn } w = v \\ \sigma(w) & \text{wenn } w \neq v \end{cases}$$

Sei $\sigma(x) = 1$ und $\sigma(w) = 0$ für $w \neq x$.

$$(x := x + 1, \sigma) \Rightarrow \sigma' \qquad \text{wobei } \sigma'(w) = \begin{cases} [x+1]\,\sigma = 2 & \text{wenn } w = x \\ \sigma(w) = 0 & \text{wenn } w \neq v \end{cases}$$

Semantik der Terme

Zuweisungen, `if`- und `while`-Anweisungen werten Terme e aus.
$[e] \colon \mathcal{S} \mapsto \mathbb{Z}$ … (partielle) Funktion, die Term e berechnet

$$
\begin{aligned}
[v]\,\sigma &= \sigma(v) && \text{wenn } v \in \mathcal{L}(Var) \\
[n]\,\sigma &= [n] && [n] \in \mathbb{Z} \ldots \text{ Zahl zum Numeral } n \in \mathcal{L}(Const) \\
[u\,e]\,\sigma &= [u]([e]\,\sigma) && [u] \colon \mathbb{Z} \mapsto \mathbb{Z} \ldots \text{ Fnkt. zum Op. } u \in \mathcal{L}(UnOp) \\
[e\,b\,e']\,\sigma &= [b]([e]\,\sigma, [e']\,\sigma) && [b] \colon \mathbb{Z}^2 \mapsto \mathbb{Z} \ldots \text{ Fnkt. zum Op. } b \in \mathcal{L}(BinOp)
\end{aligned}
$$

Wir überladen [.]!

- $[\Pi] \colon \mathcal{S} \mapsto \mathcal{S}$ … Auswertung von Programmen
- $[e] \colon \mathcal{S} \mapsto \mathbb{Z}$ … Auswertung von Termen
- $[n]$, $[u]$, $[b]$ … Zahl bzw. Funktion zum Numeral bzw. Operator

Semantik der Operatoren

Boolesche Funktionen:

$[\neg]$	
0	1
$\neq 0$	0

$[\wedge]$	0	$\neq 0$
0	0	0
$\neq 0$	0	1

$[\vee]$	0	$\neq 0$
0	0	1
$\neq 0$	1	1

$[\supset]$	0	$\neq 0$
0	1	1
$\neq 0$	0	1

Integer Funktionen:

$[+]$ ... Integer Addition oder positives Vorzeichen $(+)$
$[-]$ ... Integer Subtraktion oder negatives Vorzeichen $(-)$
$[*]$ ... Integer Multiplikation $(\cdot)$
$[/]$ ... Integer Division (mit geeigneter Rundung)

Integer Prädikate:

$[<]$, $[\leq]$, $[=]$, $[\neq]$, $[\geq]$, $[>]$ ... Vergleich von ganzen Zahlen

Rückblende

„Grundzüge digitaler Systeme“, „Formale Modellierung“

$\mathrm{val}_{I,\sigma}(t)$ ... Wert des Terms t in der Interpretation I und der Variablenbelegung σ

$\mathrm{val}_{I,\sigma}(F)$ ... Wert der prädikatenlogischen Formel F in der Interpretation I und der Variablenbelegung σ

In dieser Lehrveranstaltung: I entfällt, da die Interpretation fixiert ist (z.B. steht der binäre Operator “$+$” immer für die Addition).

$\mathrm{val}_\sigma(F)$, $\mathrm{val}(\sigma, F)$, $\mathrm{val}(F, \sigma)$, $\mathrm{val}_F(\sigma)$, $\mathrm{val}_{F,\sigma}$: verschiedene Schreibweisen für eine Funktion mit zwei Argumenten, mit unterschiedlicher Betonung der Argumente.

In dieser Lehrveranstaltung: Statt $\mathrm{val}_\sigma(F)$ schreiben wir $[F](\sigma)$ oder $[F]\sigma$, um den Zusammenhang zwischen einem Programm/Term F und der Funktion $[F]$ in den Vordergrund zu stellen. σ ist dabei der Input für $[F]$.

Beispiel: Termauswertung

Term $((x < y) \wedge (y < 2 * x + 1))$ Zustand σ: $x \mapsto 1$, $y \mapsto 2$

$$
\begin{aligned}
& [x < y \wedge y < 2 * x + 1]\,\sigma \\
& = [\wedge]([x < y]\,\sigma,\ [y < 2 * x + 1]\,\sigma) \\
& = [\wedge]([<]([x]\,\sigma, [y]\,\sigma),\ [<]([y]\,\sigma, [2 * x + 1]\,\sigma)) \\
& = [\wedge]([<](\sigma(x), \sigma(y)),\ [<](\sigma(y), [+]([2 * x]\,\sigma, [1]\,\sigma))) \\
& = [\wedge]([<](1, 2),\ [<](2, [+]([*]([2]\,\sigma, [x]\,\sigma), 1))) \\
& = [\wedge](1,\ [<](2, [+]([*](2, \sigma(x)), 1))) \\
& = [\wedge](1,\ [<](2, [+]([*](2, 1), 1))) \\
& = [\wedge](1,\ [<](2, [+](2, 1))) \\
& = [\wedge](1,\ [<](2, 3)) = [\wedge](1,\ 1) = 1
\end{aligned}
$$

Programmlauf

Programmlauf: Folge $\langle \gamma_0, \gamma_1, \gamma_2, \gamma_3, \dots \rangle$ von Konfigurationen sodass $\gamma_0 \Rightarrow \gamma_1$, $\gamma_1 \Rightarrow \gamma_2$, $\gamma_2 \Rightarrow \gamma_3$, $\dots$ gilt
(kurz: $\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \gamma_3 \Rightarrow \cdots$)

Endlicher Programmlauf der Länge k: $\gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_k$

Ein endlicher Programmlauf ist vollständig, wenn γ_k keinen Nachfolger hat, d.h., wenn es kein $\gamma \in \mathcal{C}$ mit $\gamma_k \Rightarrow \gamma$ gibt.

Die letzte Konfiguration γ_k heißt

- final, wenn $\gamma_k \in \mathcal{S}$: Das Programm terminiert.
- stecken geblieben, wenn $\gamma_k \in (\mathcal{L}(\mathit{Programm} \times \mathcal{S})$: Das Programn bricht ab.

Unendlicher Programmlauf: Das Program loopt (befindet sich in einer Endlosschleife).

Beispiel: Programmläufe

$$
\begin{aligned}
(\Pi, \sigma) &= (\{\, z := 0; \texttt{while} \ldots \}, \sigma) \\
&\quad (z := 0, \sigma) \Rightarrow \sigma_1 \\
&\Rightarrow (\texttt{while } y \neq 0 \texttt{ do} \ldots, \sigma_1) \\
&\Rightarrow (\{\{\, z := z + x; y := y - 1\}; \texttt{while} \ldots \}, \sigma_1) \\
&\quad (\{\, z := z + x; y := y - 1\}, \sigma_1) \\
&\qquad (z := z + x, \sigma_1) \Rightarrow \sigma_2 \\
&\quad \Rightarrow (y := y - 1, \sigma_2) \\
&\Rightarrow (\{\, y := y - 1; \texttt{while} \ldots \}, \sigma_2) \\
&\quad (y := y - 1, \sigma_2) \Rightarrow \sigma_3 \\
&\Rightarrow (\texttt{while } y \neq 0 \texttt{ do} \ldots, \sigma_3) \\
&\Rightarrow (\{\{\, z := z + x; y := y - 1\}; \texttt{while} \ldots \}, \sigma_3) \\
&\quad (\{\, z := z + x; y := y - 1\}, \sigma_3) \\
&\qquad (z := z + x, \sigma_1) \Rightarrow \sigma_4 \\
&\quad \Rightarrow (y := y - 1, \sigma_4) \\
&\Rightarrow (\{\, y := y - 1; \texttt{while} \ldots \}, \sigma_4) \\
&\quad (y := y - 1, \sigma_2) \Rightarrow \sigma_5 \\
&\Rightarrow (\texttt{while } y \neq 0 \texttt{ do} \ldots, \sigma_5) \\
&\Rightarrow \sigma_5
\end{aligned}
$$

Π:
```
z := 0;
while y ≠ 0 do {
  z := z + x;
  y := y - 1
}
```

$\sigma: \quad x \mapsto 3, y \mapsto 2, z \mapsto 1$

$\sigma_1: \quad z \mapsto [0]\,\sigma = 0$
$\quad x \mapsto 3, y \mapsto 2$

$[y \neq 0]\,\sigma_1 = 1$ (wahr)

$\sigma_2: \quad z \mapsto [z + x]\,\sigma_1 = 3$
$\quad x \mapsto 3, y \mapsto 2$

$\sigma_3: \quad y \mapsto [y - 1]\,\sigma_2 = 1$
$\quad x \mapsto 3, z \mapsto 3$

$[y \neq 0]\,\sigma_3 = 1$ (wahr)

$\sigma_4: \quad z \mapsto [z + x]\,\sigma_3 = 6$
$\quad x \mapsto 3, y \mapsto 1$

$\sigma_5: \quad y \mapsto [y - 1]\,\sigma_4 = 0$
$\quad x \mapsto 3, z \mapsto 6$

$[y \neq 0]\,\sigma_5 = 0$ (falsch)

Beispiel: Programmläufe

Sei $\sigma = \{x \mapsto 1\}$.

$(\{\texttt{if } x > 0 \texttt{ then abort else } \varepsilon \texttt{;} x := 2x\}, \sigma)$

$(\texttt{if } x > 0 \texttt{ then abort else } \varepsilon, \sigma)$

$[x > 0]\,\sigma = 1$ (wahr)

$\Rightarrow (\texttt{abort}, \sigma)$

$\Rightarrow (\{\texttt{abort;} x := 2x\}, \sigma)$

Der Programmlauf ist vollständig, die letzte Konfiguration bleibt stecken, das Programm bricht ab.

Beispiel: Programmläufe

$(\texttt{while 1 do } \varepsilon, \sigma)$

$\quad [1]\,\sigma = 1$ (wahr)

$\Rightarrow (\{\varepsilon\texttt{;while 1 do } \varepsilon\}, \sigma)$

$\quad (\varepsilon, \sigma) \Rightarrow \sigma$

$\Rightarrow (\texttt{while 1 do } \varepsilon, \sigma)$

$\quad [1]\,\sigma = 1$ (wahr)

$\Rightarrow \cdots$

Unendlicher Programmlauf, das Programm hängt in einer Endlosschleife.

Strukturelle operationale Semantik (SOS) von SIMPLE($\mathbb{Z}$)

Die durch Π berechnete Funktion $[\Pi] \colon \mathcal{S} \mapsto \mathcal{S}$ ist definiert als

$[\Pi]\sigma = \sigma'$ genau dann, wenn $(\Pi, \sigma) \stackrel{*}{\Rightarrow} \sigma'$ für alle Zustände $\sigma, \sigma' \in \mathcal{S}$.

($\stackrel{*}{\Rightarrow}$ ist der reflexive und transitive Abschluss von $\Rightarrow$)

Auch „small step semantics“ genannt.

Semantische Äquivalenz

Zwei Programme Π und Ω sind semantisch äquivalent, wenn $[\Pi] = [\Omega]$ gilt.

Das bedeutet:

- Wenn $(\Pi, \sigma) \stackrel{*}{\Rightarrow} \sigma'$, dann auch $(\Omega, \sigma) \stackrel{*}{\Rightarrow} \sigma'$, und umgekehrt.
- Wenn (Π, σ) in eine Endlosschleife gerät oder abbricht, dann auch (Ω, σ), und umgekehrt.

Die Semantikfunktion $[\Pi]$ unterscheidet nicht zwischen Endlosschleife und Abbruch, obwohl sich die Übergangsrelation unterschiedlich verhält.

Theorem

Für alle Programme Π, Ω, Variable v, Terme e und alle Zustände σ hat die SOS von SIMPLE($\mathbb{Z}$) folgende Eigenschaften.

- $[\varepsilon]\,\sigma = \sigma$
- $[v := e]\,\sigma = \sigma'$, wobei $\sigma'(w) = \begin{cases} [e]\,\sigma & \text{wenn } w = v \\ \sigma(w) & \text{wenn } w \neq v \end{cases}$
- $[\{\Pi\,;\Omega\}]\,\sigma = [\Omega]\,[\Pi]\,\sigma$
- $[\texttt{if } e \texttt{ then } \Pi \texttt{ else } \Omega]\,\sigma = \begin{cases} [\Pi]\,\sigma & \text{wenn } [e]\,\sigma \neq 0 \\ [\Omega]\,\sigma & \text{wenn } [e]\,\sigma = 0 \end{cases}$
- $[\texttt{while } e \texttt{ do } \Pi]\,\sigma = \begin{cases} [\texttt{while } e \texttt{ do } \Pi]\,[\Pi]\,\sigma & \text{wenn } [e]\,\sigma \neq 0 \\ \sigma & \text{wenn } [e]\,\sigma = 0 \end{cases}$

$$[\texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}\ \Omega]\,\sigma = \begin{cases} [\Pi]\,\sigma & \text{wenn } [e]\,\sigma \neq 0 \\ [\Omega]\,\sigma & \text{wenn } [e]\,\sigma = 0 \end{cases}$$

Beweis:

$[\texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}\ \Omega]\,\sigma = \sigma'$

$$\iff (\texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}\ \Omega,\ \sigma) \overset{*}{\Rightarrow} \sigma' \qquad \text{Definition von } [.]$$

$$\text{SOS: } (\texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}\ \Omega, \sigma) \Rightarrow \begin{cases} (\Pi, \sigma) & \text{wenn } [e]\,\sigma \neq 0 \\ (\Omega, \sigma) & \text{wenn } [e]\,\sigma = 0 \end{cases}$$

$$\iff \begin{cases} (\Pi, \sigma) \overset{*}{\Rightarrow} \sigma' & \text{wenn } [e]\,\sigma \neq 0 \\ (\Omega, \sigma) \overset{*}{\Rightarrow} \sigma' & \text{wenn } [e]\,\sigma = 0 \end{cases} \qquad \text{Definition von SOS für } \texttt{if}$$

$$\iff \begin{cases} [\Pi]\,\sigma = \sigma' & \text{wenn } [e]\,\sigma \neq 0 \\ [\Omega]\,\sigma = \sigma' & \text{wenn } [e]\,\sigma = 0 \end{cases} \qquad \text{Definition von } [.]$$

Hintereinanderausführung ist assoziativ

$[\{\{\Pi_1 ; \Pi_2\} ; \Pi_3\}] = [\{\Pi_1 ; \{\Pi_2 ; \Pi_3\}\}]$ für alle Programme Π_1, Π_2, and Π_3.

Beweis: Sei τ ein beliebiger Zustand. Es gilt:

$$
\begin{aligned}
[\{\{\Pi_1 ; \Pi_2\} ; \Pi_3\}]\,\tau &= [\Pi_3]\,([\{\Pi_1 ; \Pi_2\}]\,\tau) && \text{Theorem: } [\{\Pi ; \Omega\}]\,\sigma = [\Omega]\,[\Pi]\,\sigma \\
&= [\Pi_3]\,([\Pi_2]\,([\Pi_1]\,\tau)) && \text{Theorem: } [\{\Pi ; \Omega\}]\,\sigma = [\Omega]\,[\Pi]\,\sigma \\
&= [\{\Pi_2 ; \Pi_3\}]\,([\Pi_1]\,\tau) && \text{Theorem: } [\{\Pi ; \Omega\}]\,\sigma = [\Omega]\,[\Pi]\,\sigma \\
&= [\{\Pi_1 ; \{\Pi_2 ; \Pi_3\}\}]\,\tau && \text{Theorem: } [\{\Pi ; \Omega\}]\,\sigma = [\Omega]\,[\Pi]\,\sigma
\end{aligned}
$$

Dieses Resultat erlaubt uns, die erste Anweisung einer geschachtelten Hintereinanderausführung auszuführen, unabhängig von der Art der Schachtelung.

Natürliche Semantik von SIMPLE($\mathbb{Z}$)

Idee: Verwende das Theorem als Definition von $[\![\Pi]\!] : \mathcal{S} \mapsto \mathcal{S}$

Definition

Für alle Programme Π, Ω, Variable v, Terme e und alle Zustände σ ist die natürliche Semantik von SIMPLE($\mathbb{Z}$) wie folgt definiert.

- $[\![\varepsilon]\!]\,\sigma = \sigma$
- $[\![v := e]\!]\,\sigma = \sigma'$, wobei $\sigma'(w) = \begin{cases} [\![e]\!]\,\sigma & \text{wenn } w = v \\ \sigma(w) & \text{wenn } w \neq v \end{cases}$
- $[\![\{\Pi\,;\Omega\}]\!]\,\sigma = [\![\Omega]\!]\,[\![\Pi]\!]\,\sigma$
- $[\![\texttt{if } e \texttt{ then } \Pi \texttt{ else } \Omega]\!]\,\sigma = \begin{cases} [\![\Pi]\!]\,\sigma & \text{wenn } [\![e]\!]\,\sigma \neq 0 \\ [\![\Omega]\!]\,\sigma & \text{wenn } [\![e]\!]\,\sigma = 0 \end{cases}$
- $[\![\texttt{while } e \texttt{ do } \Pi]\!]\,\sigma = \begin{cases} [\![\texttt{while } e \texttt{ do } \Pi]\!]\,[\![\Pi]\!]\,\sigma & \text{wenn } [\![e]\!]\,\sigma \neq 0 \\ \sigma & \text{wenn } [\![e]\!]\,\sigma = 0 \end{cases}$

Beispiel: Natürliche Semantik

$$
\begin{aligned}
[\Pi]\,\sigma &= [\{z := 0; \texttt{while}\ \ldots\ \}]\,\sigma \\
&= [\texttt{while}\ \ldots]\,[z := 0]\,\sigma \\
&= [\texttt{while}\ y \neq 0\ \texttt{do}\ \ldots]\,\sigma_1 \\
&= [\texttt{while}\ \ldots]\,[\{z := z + x; y := y - 1\}]\,\sigma_1 \\
&= [\texttt{while}\ \ldots]\,[y := y - 1]\,[z := z + x]\,\sigma_1 \\
&= [\texttt{while}\ \ldots]\,[y := y - 1]\,\sigma_2 \\
&= [\texttt{while}\ y \neq 0\ \texttt{do}\ \ldots]\,\sigma_3 \\
&= [\texttt{while}\ \ldots]\,[\{z := z + x; y := y - 1\}]\,\sigma_3 \\
&= [\texttt{while}\ \ldots]\,[y := y - 1]\,[z := z + x]\,\sigma_3 \\
&= [\texttt{while}\ \ldots]\,[y := y - 1]\,\sigma_4 \\
&= [\texttt{while}\ y \neq 0\ \texttt{do}\ \ldots]\,\sigma_5 \\
&= \sigma_5
\end{aligned}
$$

Π:
```
z := 0;
while y ≠ 0 do {
  z := z + x;
  y := y - 1
}
```

$\sigma: \quad x \mapsto 3, y \mapsto 2, z \mapsto 1$

$\sigma_1: \quad z \mapsto [0]\,\sigma = 0$
$\quad x \mapsto 3, y \mapsto 2$

$[y \neq 0]\,\sigma_1 = 1$ (wahr)

$\sigma_2: \quad z \mapsto [z + x]\,\sigma_1 = 3$
$\quad x \mapsto 3, y \mapsto 2$

$\sigma_3: \quad y \mapsto [y - 1]\,\sigma_2 = 1$
$\quad x \mapsto 3, z \mapsto 3$

$[y \neq 0]\,\sigma_3 = 1$ (wahr)

$\sigma_4: \quad z \mapsto [z + x]\,\sigma_3 = 6$
$\quad x \mapsto 3, y \mapsto 1$

$\sigma_5: \quad y \mapsto [y - 1]\,\sigma_4 = 0$
$\quad x \mapsto 3, z \mapsto 6$

$[y \neq 0]\,\sigma_5 = 0$ (falsch)

Vergleich mit SOS

$$
\begin{aligned}
(\Pi, \sigma) &= (\{\, z := 0; \texttt{while} \ldots \}, \sigma) \\
&\quad (z := 0, \sigma) \Rightarrow \sigma_1 \\
&\Rightarrow (\texttt{while } y \neq 0 \texttt{ do} \ldots, \sigma_1) \\
&\Rightarrow (\{\{\, z := z + x; y := y - 1\}; \texttt{while} \ldots \}, \sigma_1) \\
&\quad (\{\, z := z + x; y := y - 1\}, \sigma_1) \\
&\qquad (z := z + x, \sigma_1) \Rightarrow \sigma_2 \\
&\quad \Rightarrow (y := y - 1, \sigma_2) \\
&\Rightarrow (\{\, y := y - 1; \texttt{while} \ldots \}, \sigma_2) \\
&\quad (y := y - 1, \sigma_2) \Rightarrow \sigma_3 \\
&\Rightarrow (\texttt{while } y \neq 0 \texttt{ do} \ldots, \sigma_3) \\
&\Rightarrow (\{\{\, z := z + x; y := y - 1\}; \texttt{while} \ldots \}, \sigma_3) \\
&\quad (\{\, z := z + x; y := y - 1\}, \sigma_3) \\
&\qquad (z := z + x, \sigma_1) \Rightarrow \sigma_4 \\
&\quad \Rightarrow (y := y - 1, \sigma_4) \\
&\Rightarrow (\{\, y := y - 1; \texttt{while} \ldots \}, \sigma_4) \\
&\quad (y := y - 1, \sigma_2) \Rightarrow \sigma_5 \\
&\Rightarrow (\texttt{while } y \neq 0 \texttt{ do} \ldots, \sigma_5) \\
&\Rightarrow \sigma_5
\end{aligned}
$$

Π:
```
z := 0;
while y ≠ 0 do {
  z := z + x;
  y := y - 1
}
```

$\sigma: \quad x \mapsto 3, y \mapsto 2, z \mapsto 1$

$\sigma_1: \quad z \mapsto [0]\,\sigma = 0$
$\qquad x \mapsto 3, y \mapsto 2$

$[y \neq 0]\,\sigma_1 = 1$ (wahr)

$\sigma_2: \quad z \mapsto [z + x]\,\sigma_1 = 3$
$\qquad x \mapsto 3, y \mapsto 2$

$\sigma_3: \quad y \mapsto [y - 1]\,\sigma_2 = 1$
$\qquad x \mapsto 3, z \mapsto 3$

$[y \neq 0]\,\sigma_3 = 1$ (wahr)

$\sigma_4: \quad z \mapsto [z + x]\,\sigma_3 = 6$
$\qquad x \mapsto 3, y \mapsto 1$

$\sigma_5: \quad y \mapsto [y - 1]\,\sigma_4 = 0$
$\qquad x \mapsto 3, z \mapsto 6$

$[y \neq 0]\,\sigma_5 = 0$ (falsch)

Strukturelle operationale versus natürliche Semantik

Natürliche Semantik (natural semantics):

- keine Übergänge, keine Programmläufe
- eine rekursive Definition, die Ein- zu Ausgangszustände in Beziehung setzt
- eleganter, einfacher zu verwenden, kompakter

Strukturelle operationale Semantik (structural operational semantics, small step semantics):

- näher an der realen Maschine, da die einzelne Schritte modelliert werden
- kann zwischen Endlosschleifen und Abbrüchen unterscheiden
- würde es z.B. erlauben, low-level Parallelismus korrekt zu modellieren
- ist flexibler und detailreicher

Ist dieses Programm korrekt?

```
// Multipliziert x und y
z := 0;
while y ≠ 0 do {
   z := z + x;
   y := y − 1
}
```

- ✓ Ist das ein zulässiges Programm? Was alles ist zulässig? ⇒ Syntax
- ✓ Was bedeuten die Programmelemente? ⇒ operationale Semantik
- Wie ist die Behauptung „Multipliziert x und y" zu interpretieren?
  ⇒ Korrektheitsaussagen
- Wie können wir so eine Behauptung (automatisch) beweisen?
  ⇒ axiomatische Semantik (Hoare-Kalkül)

Spezifikation

- Beschreibung der zulässigen Eingaben (Vorbedingung, Precondition)
- Beschreibung des Resultats (Nachbedingung, Postcondition)

Spezifikationssprachen:

Java	$\Rightarrow$	JML (Java Modelling Language), basiert auf Prädikatenlogik
C#	$\Rightarrow$	Spec#, basiert auf Prädikatenlogik
	$\vdots$	
SIMPLE($\mathbb{Z}$)	$\Rightarrow$	Prädikatenlogik

Eine Formel repräsentiert alle Zustände, in denen sie wahr ist.
Eine Formel F repräsentiert die Menge $\{\sigma \in \mathcal{S} \mid [F]\,\sigma = \text{wahr}\}$.

$\exists i\,(x = 2i)$

repräsentiert alle Zustände σ, in denen $\sigma(x)$ eine gerade Zahl ist.

Prädikatenlogische Formeln

Syntax

Formel $\rightarrow$ *Var* | *Const* | *UnOp Formel* | "(" *Formel BinOp Formel* ")"
| "$\forall$" *Var Formel* | "$\exists$" *Var Formel*

Semantik = Semantik der Terme sowie

$$[\forall v\, F]\,\sigma = \begin{cases} 1 & \text{wenn } [F]\,\sigma' \neq 0 \text{ (wahr) für alle } \sigma' \in \mathcal{S} \text{ mit } \sigma' \overset{v}{\sim} \sigma \\ 0 & \text{sonst} \end{cases}$$

$$[\exists v\, F]\,\sigma = \begin{cases} 1 & \text{wenn } [F]\,\sigma' \neq 0 \text{ (wahr) für mind. ein } \sigma' \in \mathcal{S} \text{ mit } \sigma' \overset{v}{\sim} \sigma \\ 0 & \text{sonst} \end{cases}$$

v ... die durch den Quantor gebundene Variable

$\sigma' \overset{v}{\sim} \sigma$: $\sigma'(x) = \sigma(x)$ für alle Variablen $x \neq v$

"Die Zustände σ and σ' unterscheiden sich höchstens in der Variablen v."

Warnhinweis nach dem Konsumentenschutzgesetz

Diese Definition der Prädikatenlogik kann Ihr geistiges Wohlbefinden beeinträchtigen!

Erlaubt Unsinnigkeiten, die Ihnen in GDS oder FMOD Punkte gekostet hätten.

Schachtelung von Prädikatensymbolen

$((1 < 2) < 3)$ ist in ThInf zulässig, da wir $\mathbb{Z}$ auch für die Wahrheitswerte verwenden.
$[((1 < 2) < 3)]\,\sigma = [<]\,([<]\,(1, 2), 3) = [<]\,(1, 3) = 1$

Terme als Formeln

$[\forall x\, x]\,\sigma = 0$ für beliebiges σ.
$[\exists x\, x]\,\sigma = 1$ für beliebiges σ.

Korrektheitsaussagen

Korrektheitsaussage = Programm + Spezifikation

{ Vorbedingung }	Programm	{ Nachbedingung }
{ PL-Formel }	SIMPLE($\mathbb{Z}$)-Programm	{ PL-Formel }
$\{ F \}$	Π	$\{ G \}$

Vorbedingung $\{ x = x_0 \wedge y = y_0 \}$

```
z := 0;
while y ≠ 0 do {
   z := z + x;
   y := y − 1
}
```

Nachbedingung $\{ z = x_0 * y_0 \}$

x_0, y_0 ... „logische" Variablen, treten nicht im Programm auf, haben selben Wert in Vor- und Nachbedingung.

Wann ist eine Korrektheitsaussage $\{F\}\,\Pi\,\{G\}$ wahr?

Jede zulässige Eingabe führt zu einem richtigen Ergebnis.

Für jeden Zustand gilt:
Wenn er die Vorbedingung erfüllt,
dann erfüllt der Zustand bei Programmende die Nachbedingung.

Für alle $\sigma \in \mathcal{S}$ gilt:
Wenn $[F]\,\sigma$ wahr ist,
dann ist auch $[G]\,\sigma'$ wahr, wobei $\sigma' = [\Pi]\,\sigma$.

Was passiert, wenn das Programm für manche Eingaben nicht terminiert?

2 Möglichkeiten = 2 Korrektheitsbegriffe:

- Partielle Korrektheit:
  Falls die Eingabe zulässig ist und falls das Programm terminiert,
  dann ist das Ergebnis richtig.
- Totale Korrektheit:
  Falls die Eingabe zulässig ist,
  dann terminiert das Programm und das Ergebnis ist richtig.

$\{F\}\,\Pi\,\{G\}$ ist wahr bzgl. partieller Korrektheit

Für alle $\sigma \in \mathcal{S}$ gilt:
Wenn $[F]\,\sigma$ wahr ist und $\sigma' = [\Pi]\,\sigma$ definiert ist,
dann ist $[G]\,\sigma'$ wahr.

„$\{F\}\,\Pi\,\{G\}$ ist partiell korrekt.“

$\{F\}\,\Pi\,\{G\}$ ist wahr bzgl. totaler Korrektheit

Für alle $\sigma \in \mathcal{S}$ gilt:
Wenn $[F]\,\sigma$ wahr ist,
dann ist $\sigma' = [\Pi]\,\sigma$ definiert und $[G]\,\sigma'$ ist wahr.

„$\{F\}\,\Pi\,\{G\}$ ist total korrekt.“

Totale Korrektheit $=$ Partielle Korrektheit $+$ Termination

Wahr oder falsch, partiell oder total korrekt?

$\{1\}\, x := 2\, \{x = 2\}$

Total (und daher auch partiell) korrekt.

$\{1\}\, x := 2\, \{x = 3\}$

Weder total noch partiell korrekt. Gegenbeispiel: beliebig

$\{1\}\, \texttt{while}\ x > 2\ \texttt{do}\ x := x - 1\, \{x = 2\}$

Weder total noch partiell korrekt. Gegenbeispiel: $\sigma(x) = 0$

$\{1\}\, \texttt{while}\ x \neq 2\ \texttt{do}\ x := x - 1\, \{x = 2\}$

Partiell aber nicht total korrekt. Gegenbeispiel: $\sigma(x) = 0$

$\{x > 5\}\, \texttt{while}\ x \neq 2\ \texttt{do}\ x := x - 1\, \{x = 2\}$

Total (und daher auch partiell) korrekt.

$\{\, x = x_0 \land y = y_0 \,\}$

```
z := 0;
while y ≠ 0 do {
  z := z + x;
  y := y − 1
}
```

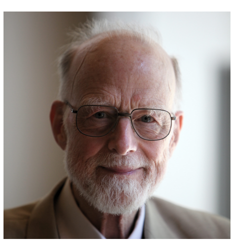
$\{\, z = x_0 * y_0 \,\}$

Partiell korrekt? Scheint so.

Terminiert das Programm? Nein, nicht für $y < 0$.
$\Longrightarrow$ Erweitere die Vorbedingung um das Konjunkt $y \geq 0$.

Aber wie lässt sich das systematisch beweisen?
Es sind unendlich viele Wertebelegungen zu überprüfen, nicht machbar.
$\Longrightarrow$ Hoare-Kalkül

Tony Hoare



Sir Charles Antony Richard Hoare, besser bekannt als Tony Hoare (geb. 1934). Britischer Informatiker, Turing-Award-Preisträger. Bekannt für die Erfindung von Quicksort, die Entwicklung des Hoare-Kalküls zur Programmverifikation und der Spezifikationssprache „Communicating Sequential Processes" (CSP).

Meilenstein: An axiomatic basis for computer programming (CACM, 1969)

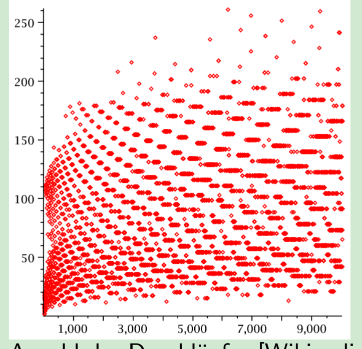
> There are two ways of constructing a software design.
> One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.
> The first method is far more difficult.

> The unavoidable price of reliability is simplicity.

$\{ x \geq 1 \}$

```
while x > 1 do
  if x = (x/2) * 2 then
    x := x/2
  else
    x := 3 * x + 1
```

$\{ x = 1 \}$



Anzahl der Durchläufe [Wikipedia]

Partiell korrekt? Offensichtlich.

x ist immer positiv.

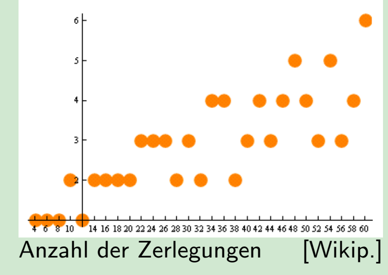
Wenn das Programm terminiert, gilt $x \not> 1$, daher muss x gleich 1 sein.

Terminiert das Programm? Ja, falls $x_n = 1$ für ein n, für alle $x_0 \geq 1$.

$$x_{n+1} = \begin{cases} x_n/2 & \text{wenn } x_n \text{ gerade} \\ 3x_n + 1 & \text{wenn } x_n \text{ ungerade} \end{cases}$$

Collatz Vermutung, offenes Problem

$\{ x \geq 2 \}$
$y := 2;$
$\texttt{while } y < x$
$\quad \wedge \neg(\text{IstPrim}(y) \wedge \text{IstPrim}(2x - y)) \texttt{ do}$
$\quad y := y + 1$
$\{ \text{IstPrim}(y) \wedge \text{IstPrim}(2x - y) \}$



Anzahl der Zerlegungen [Wikip.]

Partiell korrekt?
Ja, falls sich jede gerade ganze Zahl größer als 2 als Summe zweier Primzahlen darstellen lässt.
Goldbach Vermutung, offenes Problem.

Terminiert das Programm? Offensichtlich, höchstens $x - 2$ Durchläufe.

- Manche Zusicherungen sind für jede Verifikationmethode knifflig.
- In der Praxis sind die Verifikationsaufgaben viel einfacher.

Ist dieses Programm korrekt?

```
// Multipliziert x und y
z := 0;
while y ≠ 0 do {
   z := z + x;
   y := y − 1
}
```

- ✓ Ist das ein zulässiges Programm? Was alles ist zulässig? $\Rightarrow$ Syntax
- ✓ Was bedeuten die Programmelemente? $\Rightarrow$ operationale Semantik
- ✓ Wie ist die Behauptung „Multipliziert x und y“ zu interpretieren? $\Rightarrow$ Korrektheitsaussagen
- Wie können wir so eine Behauptung (automatisch) beweisen? $\Rightarrow$ axiomatische Semantik (Hoare-Kalkül)

Der Hoare-Kalkül

Sammlung von Regeln, die Korrektheitsaussagen in einfachere zerlegen

... bis „nur“ noch prädikatenlog. Formeln bleiben, deren Gültigkeit zu zeigen ist.

Korrektheit des Hoare-Kalküls

Jede mit dem Hoare-Kalkül bewiesene Korrektheitsaussage ist wahr.

Vollständigkeit des Hoare-Kalküls

Jede wahre Korrektheitsaussage lässt sich mit dem Hoare-Kalkül beweisen (falls die Gültigkeit der auftretenden Formeln bewiesen werden kann).

Genau genommen sind es zwei (ähnliche) Kalküle, einer für partielle Korrektheit und einer für totale Korrektheit.

Schreibweise für Regeln:

$$\frac{F_1 \supset G_1 \quad \{F_2\}\,\Pi_2\,\{G_2\} \quad F_3 \supset G_3 \quad \{F_4\}\,\Pi_4\,\{G_4\} \quad \cdots}{\{F\}\,\Pi\,\{G\}} \qquad \begin{array}{l}\text{Prämissen}\\ \text{Konklusion}\end{array}$$

Korrektheit: Immer, wenn alle Prämissen gültig (im Fall von Formeln) bzw. wahr (im Fall von Korrektheitsaussagen) sind, ist die Konklusion wahr.

Implikationsregel

Wenn $F \supset F'$ und $G' \supset G$ gültige Formeln sind
und $\{F'\}\,\Pi\,\{G'\}$ eine wahre Korrektheitsaussage ist,
dann ist auch $\{F\}\,\Pi\,\{G\}$ eine wahre Korrektheitsaussage.

$$\frac{F \supset F' \quad \{F'\}\,\Pi\,\{G'\} \quad G' \supset G}{\{F\}\,\Pi\,\{G\}}\,\text{(imp)}$$

$$\frac{x > 5 \supset x > 0 \quad \{x > 0\}\,x := 2\,\{x = 2\} \quad x = 2 \supset x \leq 3}{\{x > 5\}\,x := 2\,\{x \leq 3\}}\,\text{(imp)}$$

Axiome

Spezialfall von Regeln ohne Prämissen

Schreibweise für Axiome:

$$\frac{1}{\{F\}\,\Pi\,\{G\}} \quad \text{oder} \quad \frac{}{\{F\}\,\Pi\,\{G\}} \quad \text{oder} \quad \{F\}\,\Pi\,\{G\}$$

Korrektheit: Jede Korrektheitsaussage der Form $\{F\}\,\Pi\,\{G\}$ ist wahr.

$\{F\}\ v := v\ \{F\}$ ist ein korrektes Axiom.

Jede Instanz ist eine wahre Korrektheitsaussage:

$\{x > 2\}\ x := x\ \{x > 2\}$

$\{1\}\ z := z\ \{1\}$

$\{x = 0 \land y = z\}\ z := z\ \{x = 0 \land y = z\}$

...

Zuweisungen (1)

$\{F\}\, v := e\, \{?\}$

$\{x = 2\}\, y := 3\, \{x = 2 \wedge y = 3\}$ ✓

Verallgemeinerung dieser Beobachtung:
$\{F\}\, v := e\, \{F \wedge v = e\}$

$\{x = 2\}\, x := 3\, \{x = 2 \wedge x = 3\}$???
$\Rightarrow$ v darf nicht in F vorkommen!

$\{x = 2\}\, y := y + 1\, \{x = 2 \wedge y = y + 1\}$???
$\Rightarrow$ v darf nicht in e vorkommen!

$\{F\}\, v := e\, \{F \wedge v = e\}$ korrekt, wenn v weder in F noch in e vorkommt.

Nützlich für die Initialisierung lokaler Variablen.
Aber was tun mit den anderen Zuweisungen?

Zuweisungen (2)

$$\{x = 2\} \qquad x := x + 5 \qquad \{x - 5 = 2\}$$

Wenn $\sigma(x) = 2$ und $\sigma'(x) = \sigma(x) + 5$ dann $\sigma'(x) - 5 = 2$.

$$\sigma'(x) - 5 = \sigma(x)$$

Verallgemeinerung der Beobachtung:
$\{F\}\, v := e(v)\, \{F[{}^{e^{-1}(v)}_{v}]\}$

e^{-1} ... 'inverser' Ausdruck zu e
$F[{}^{e^{-1}(v)}_{v}]$ ... Formel F, in der jedes v durch $e^{-1}(v)$ ersetzt wurde (nach Umbenennung gebundener Variablen)

$\{y = 4x\}\, x := 2x\, \{y = 4(x/2)\}$ $\quad e(x) = 2x \quad e^{-1}(x) = x/2$
$\{y = 4x\}\, x := 2x\, \{y = 2x\}$ ✓

$\{y = 4x\}\, x := x/2\, \{\qquad\}$

Was ist die inverse Funktion zu $x/2$? Ist es $2x$? Oder $2x + 1$?
Was, wenn die Inverse nicht explizit ausgedrückt werden kann?
Was ist die Inverse einer Konstanten?

Zuweisung (3)

$\{F\}\, v := e\, \{?\}$

Was wissen wir über den Zustand nach der Zuweisung?

- Es gibt einen alten Wert von v (nicht mehr verfügbar). $\exists v'$
- F is wahr, aber für den alten Wert von v. $F\left[{v' \atop v}\right]$
- Der neue Wert von v ist der Wert von e (aber berechnet mit dem alten Wert von v). $v = e\left[{v' \atop v}\right]$

$\{F\}\, v := e\, \{\exists v'\, (F\left[{v' \atop v}\right] \wedge v = e\left[{v' \atop v}\right])\}$ ... korrektes Axiom

(v' ist eine neue Variable, die nicht in F und e vorkommt.)

$$\{y = 4x\}\, x := 2x\, \{\exists x'\, (y = 4x' \wedge x = 2x')\}$$
$$\exists x'\, (y = 2x \wedge x = 2x')$$
$$y = 2x \wedge \exists x'\, x = 2x'$$
$$y = 2x \wedge x \text{ ist gerade}$$

$$\{y = 4x\}\, x := x/2\, \{\exists x'\, (y = 4x' \wedge x = x'/2)\}$$

Zuweisung (4)

$$\{x + 5 = 7\} \qquad x := x + 5 \qquad \{x = 7\}$$

Wenn $\sigma(x) + 5 = 7$ und $\sigma'(x) = \sigma(x) + 5$ dann $\sigma'(x) = 7$.

Verallgemeinerung der Beobachtung:
$\{G\left[{e \atop v}\right]\}\; v := e \;\{G\}$ ... korrektes Axiom

$G\left[{e \atop v}\right]$ ... Formel, die aus G in zwei Schritten entsteht:

1. Benenne alle gebundenen Variablen in G so um, dass weder v noch die Variablen in e gebunden in G vorkommen.
2. Ersetze in G alle v durch e.

$\{y = 2(2x)\}\; x := 2x \;\{y = 2x\}$ ✓

$\{y = 8(x/2)\}\; x := x/2 \;\{y = 8x\}$ ✓

$$\frac{x = 2 \supset 3 = 3 \qquad \{3 = 3\}\; x := 3 \;\{x = 3\}}{\{x = 2\}\; x := 3 \;\{x = 3\}}\ \text{(imp)}$$ ✓

Axiome für die Zuweisung

$$\{G\left[{e \atop v}\right]\}\; v := e\; \{G\} \;{\scriptstyle (zw)}$$

$$\{F\}\; v := e\; \{\exists v' \left(F\left[{v' \atop v}\right] \wedge v = e\left[{v' \atop v}\right]\right)\} \;{\scriptstyle (zw')}$$

falls v' nicht in F und e vorkommt.

$$\{F\}\; v := e\; \{F \wedge v = e\} \;{\scriptstyle (zw')}$$

falls v nicht in F und e vorkommt.

Hintereinanderausführung

Angenommen $\{F\}\,\Pi\,\{G\}$ und $\{G\}\,\Omega\,\{H\}$ sind beide wahr.

Dann gilt für jede Wahrheitsbelegung σ:

- Aus $[F]\,\sigma = \text{wahr}$ folgt $[G]\,\sigma' = \text{wahr}$, wobei $\sigma' = [\Pi]\,\sigma$.
- Aus $[G]\,\sigma' = \text{wahr}$ folgt $[H]\,\sigma'' = \text{wahr}$, wobei $\sigma'' = [\Omega]\,\sigma'$.

Zusammengenommen ergibt sich:

- Aus $[F]\,\sigma = \text{wahr}$ folgt $[H]\,\sigma'' = \text{wahr}$,
  wobei $\sigma'' = [\Omega]\,\sigma' = [\Omega]\,([\Pi]\,\sigma) = [\{\Pi\,;\Omega\}]\,\sigma$.

Somit ist die Korrektheitsaussage $\{F\}\,\{\Pi\,;\Omega\}\,\{H\}$ wahr.

$$\frac{\{F\}\,\Pi\,\{G\} \qquad \{G\}\,\Omega\,\{H\}}{\{F\}\,\{\,\Pi\,;\Omega\,\}\,\{H\}}\;\text{(ha)}$$

Problem: Wie lässt sich die Interpolante G bestimmen?

Wertetausch zweier Variablen

$$\dfrac{\dfrac{F \supset G[^y_x] \quad \{G[^y_x]\}\, x := y\,\{G\}\;(\text{zw})}{\{F\}\, x := y\,\{G\}}\,(\text{imp}) \quad \dfrac{G \supset Q[^t_y] \quad \{Q[^t_y]\}\, y := t\,\{Q\}\;(\text{zw})}{\{G\}\, y := t\,\{Q\}}\,(\text{imp})}{\{F\}\,\{\; x := y; y := t\;\}\,\{Q\}}\,(\text{ha})$$

$$\dfrac{\dfrac{P \supset F[^x_t] \quad \{F[^x_t]\}\, t := x\,\{F\}\;(\text{zw})}{\{P\}\, t := x\,\{F\}}\,(\text{imp}) \qquad \{F\}\,\{\; x := y; y := t\;\}\,\{Q\}}{\{P\colon x = x_0 \land y = y_0\}\,\{\; t := x; \{\; x := y; y := t\;\}\;\}\,\{Q\colon x = y_0 \land y = x_0\}}\,(\text{ha})$$

Zu tun: Finde F und G, sodass die Formeln $P \supset F[^x_t]$, $F \supset G[^y_x]$, und $G \supset Q[^t_y]$ gültig werden.

Wir wählen $F \equiv G[^y_x]$ und $G \equiv Q[^t_y]$.

Es bleibt $P \supset F[^x_t]$ zu zeigen, d.h., $P \supset Q[^t_y][^y_x][^x_t]$.

$$P \supset Q[{}^{t}_{y}]\,[{}^{y}_{x}]\,[{}^{x}_{t}]$$

$$(x = x_0 \wedge y = y_0) \supset (x = y_0 \wedge y = x_0)[{}^{t}_{y}]\,[{}^{y}_{x}]\,[{}^{x}_{t}]$$

$$(x = x_0 \wedge y = y_0) \supset (x = y_0 \wedge t = x_0)[{}^{y}_{x}]\,[{}^{x}_{t}]$$

$$(x = x_0 \wedge y = y_0) \supset (y = y_0 \wedge t = x_0)[{}^{x}_{t}]$$

$$(x = x_0 \wedge y = y_0) \supset (y = y_0 \wedge x = x_0)$$

Diese Implikation ist gültig, daher ist die Korrektheitsaussage

$$\{P\}\,\{\ t := x; \{\ x := y; y := t\ \}\ \}\,\{Q\}$$

wahr hinsichtlich partieller bzw. totaler Korrektheit.

Wertetausch (vereinfacht)

$$
\dfrac{\dfrac{\dfrac{P \supset F\left[{x \atop t}\right] \quad \overset{\text{(zw)}}{\{F\left[{x \atop t}\right]\}\, t := x\, \{F\}}}{\{P\}\, t := x\, \{F\}}\,\text{(imp)} \qquad \overset{\text{Wähle } F = G\left[{y \atop x}\right]}{\overset{\text{(zw)}}{\{F\}\, x := y\, \{G\}}}}{\{P\}\, \{\ t := x; x := y\ \}\, \{G\}}\,\text{(ha)} \qquad \overset{\text{Wähle } G = Q\left[{t \atop y}\right]}{\overset{\text{(zw)}}{\{G\}\, y := t\, \{Q\}}}}{\{P\colon x = x_0 \wedge y = y_0\}\, \{\ \{\ t := x; x := y\ \}; y := t\ \}\, \{Q\colon x = y_0 \wedge y = x_0\}}\,\text{(ha)}
$$

- Zeige $P \supset F\left[{x \atop t}\right]$

  $P \supset G\left[{y \atop x}\right]\left[{x \atop t}\right]$

  $P \supset Q\left[{t \atop y}\right]\left[{y \atop x}\right]\left[{x \atop t}\right]$

  (Haben wir bereits vorhin gezeigt.)

Wertetausch (Beweis mit Annotationen)

$$\{\, P\colon x = x_0 \wedge y = y_0 \,\}$$
$$\{\, F_3\colon Q\left[{}^{t}_{y}\right]\left[{}^{y}_{x}\right]\left[{}^{x}_{t}\right] \,\} \quad \text{zw} \uparrow$$
$$t := x\,;$$
$$\{\, F_2\colon Q\left[{}^{t}_{y}\right]\left[{}^{y}_{x}\right] \,\} \quad \text{zw} \uparrow$$
$$x := y\,;$$
$$\{\, F_1\colon Q\left[{}^{t}_{y}\right] \,\} \quad \text{zw} \uparrow$$
$$y := t$$
$$\{\, Q\colon x = y_0 \wedge y = x_0 \,\}$$

Implikationsregel: Beweise die Implikation

$$P \supset F_3\colon \quad (x = x_0 \wedge y = y_0) \supset Q\left[{}^{t}_{y}\right]\left[{}^{y}_{x}\right]\left[{}^{x}_{t}\right]$$

Annotierungsregeln bisher

Zuweisungen

$$\begin{array}{lll} & & \{F[{}^{e}_{v}]\} \quad \text{zw}\uparrow \\ v := e & \mapsto & v := e \\ \{F\} & & \{F\} \end{array}$$

$$\begin{array}{lll} \{F\} & & \{F\} \\ v := e & \mapsto & v := e \\ & & \{\exists v'(F[{}^{v'}_{v}] \wedge v = e[{}^{v'}_{v}])\} \quad \text{zw}\downarrow \end{array}$$

$$\begin{array}{lll} \{F\} & & \{F\} \\ v := e & \mapsto & v := e \qquad \text{(falls } v \text{ nicht frei in } F \text{ und } e\text{)} \\ & & \{F \wedge v = e\} \quad \text{zw}\downarrow \end{array}$$

Annotierungsregeln bisher

Implikationsregel

$$\begin{array}{l}\{F\}\\\{G\}\end{array} \quad\mapsto\quad \begin{array}{l}\{F\}\\\{G\}\end{array} \quad \text{Zeige } F \supset G \quad \text{imp}$$

Hintereinanderausführungsregel wird implizit angewendet

$$\begin{array}{l}\\;\\\{F\}\end{array} \quad\mapsto\quad \begin{array}{ll}\{F\} & \text{sp}\uparrow\\;\\\{F\}\end{array} \qquad\qquad \begin{array}{l}\{F\}\\;\\\ \end{array} \quad\mapsto\quad \begin{array}{ll}\{F\}\\;\\\{F\} & \text{sp}\downarrow\end{array}$$

$$\begin{array}{l}\\\{\\\{F\}\end{array} \quad\mapsto\quad \begin{array}{ll}\{F\} & \text{begin}\uparrow\\\{\\\{F\}\end{array} \qquad\qquad \begin{array}{l}\{F\}\\\{\\\ \end{array} \quad\mapsto\quad \begin{array}{ll}\{F\}\\\{\\\{F\} & \text{begin}\downarrow\end{array}$$

$$\begin{array}{l}\\\}\\\{F\}\end{array} \quad\mapsto\quad \begin{array}{ll}\{F\} & \text{end}\uparrow\\\}\\\{F\}\end{array} \qquad\qquad \begin{array}{l}\{F\}\\\}\\\ \end{array} \quad\mapsto\quad \begin{array}{ll}\{F\}\\\}\\\{F\} & \text{end}\downarrow\end{array}$$

Leerprogramme

Hoare-Kalkül

$$\{F\}\ \varepsilon\ \{F\} \quad \text{(leer)}$$

Annotierungsregel

$$\begin{array}{lll} & & \{F\} \quad \text{leer} \uparrow \\ \varepsilon & \mapsto & \varepsilon \\ \{F\} & & \{F\} \end{array} \qquad \begin{array}{lll} \{F\} & & \{F\} \\ \varepsilon & \mapsto & \varepsilon \\ & & \{F\} \quad \text{leer} \downarrow \end{array}$$

Konditionale

$$\frac{\{F \wedge e\}\,\Pi\,\{G\} \quad \{F \wedge \neg e\}\,\Omega\,\{G\}}{\{F\}\,\texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}\ \Omega\,\{G\}}\ \text{(if)}$$

$$\frac{\{F\}\,\Pi\,\{H\} \quad \{G\}\,\Omega\,\{H\}}{\{(e \supset F) \wedge (\neg e \supset G)\}\,\texttt{if}\ e\ \texttt{then}\ \Pi\ \texttt{else}\ \Omega\,\{H\}}\ \text{(if}'\text{)}$$

Es genügt eine der beiden Regeln für einen vollständigen Kalkül.

Annotierungsregeln für Konditionale

$$
\begin{array}{l}
\{F\} \\
\texttt{if } e \texttt{ then} \\
\\
\quad \ldots \\
\texttt{else}
\end{array}
\quad \mapsto \quad
\begin{array}{l}
\{F\} \\
\texttt{if } e \texttt{ then} \\
\quad \{F \wedge e\} \quad \text{if} \downarrow \\
\quad \ldots \\
\texttt{else} \\
\quad \{F \wedge \neg e\} \quad \text{if} \downarrow
\end{array}
$$

$$
\begin{array}{l}
\\
\texttt{if } e \texttt{ then} \\
\quad \{F\} \\
\quad \ldots \\
\texttt{else} \\
\quad \{G\}
\end{array}
\quad \mapsto \quad
\begin{array}{l}
\{(e \supset F) \wedge (\neg e \supset G)\} \quad \text{if} \uparrow \\
\texttt{if } e \texttt{ then} \\
\quad \{F\} \\
\quad \ldots \\
\texttt{else} \\
\quad \{G\}
\end{array}
$$

Annotierungsregeln für Konditionale

```
else                     {G}  endif ↑
                         else
  ...          ↦           ...
                         {G}  endif ↑
// end if                // end if
{G}                      {G}
```

```
 {G1}                    {G1}
else                     else
  ...          ↦           ...
 {G2}                    {G2}
// end if                // end if
                         {G1 ∨ G2}  endif ↓
```

Maximum zweier Zahlen (1)

$\{ 1 \}$
`if` $x \leq y$ `then`
  $\{ 1 \wedge x \leq y \}$ if $\downarrow$
  $\{ y = \max(x, y) \}$ zw $\uparrow$
  $z := y$
  $\{ z = \max(x, y) \}$ endif $\uparrow$
`else`
  $\{ 1 \wedge x \not\leq y \}$ if $\downarrow$
  $\{ x = \max(x, y) \}$ zw $\uparrow$
  $z := x$
  $\{ z = \max(x, y) \}$ endif $\uparrow$
`// end if`
$\{ z = \max(x, y) \}$

Zu zeigen: Die Formeln

$$1 \wedge x \leq y \supset y = \max(x, y) \quad \text{(imp)}$$

$$1 \wedge x \not\leq y \supset x = \max(x, y) \quad \text{(imp)}$$

sind gültig.

Maximum zweier Zahlen (2)

$\{ 1 \}$

$\{ (x \leq y \supset y = \max(x,y)) \wedge (x \not\leq y \supset x = \max(x,y)) \}$ if $\uparrow$

if $x \leq y$ **then**

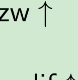
  $\{ y = \max(x,y) \}$ zw $\uparrow$

  $z := y$

  $\{ z = \max(x,y) \}$ endif $\uparrow$

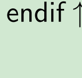
else

  $\{ x = \max(x,y) \}$ zw $\uparrow$



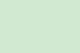
  $z := x$

  $\{ z = \max(x,y) \}$ endif $\uparrow$



// end **if**

$\{ z = \max(x,y) \}$



Zu zeigen: Die Implikation

$$1 \supset ((x \leq y \supset y = \max(x,y)) \wedge (x \not\leq y \supset x = \max(x,y)))$$

ist gültig. (Implikationsregel)

$\{am{\leq}n{<}bm\}\ d{:=}(a{+}b)/2; \texttt{if}\ dm{\leq}n\ \texttt{then}\ a{:=}d\ \texttt{else}\ b{:=}d\ \{am{\leq}n{<}bm\}$

$$F_a = F[{}^d_a] \qquad\qquad F_b = F[{}^d_b]$$

$$\dfrac{\dfrac{\overset{(2)}{G \wedge dm{\leq}n \supset F_a} \quad \overset{(zw)}{\{F_a\}\, a := d\, \{F\}}}{\{G \wedge dm \leq n\}\, a := d\, \{F\}}\text{(imp)} \quad \dfrac{\overset{(3)}{G \wedge dm{\not\leq}n \supset F_b} \quad \overset{(zw)}{\{F_b\}\, b := d\, \{F\}}}{\{G \wedge dm \not\leq n\}\, b := d\, \{F\}}\text{(imp)}}{\{G\}\, \texttt{if}\ dm \leq n\ \texttt{then}\ a := d\ \texttt{else}\ b := d\, \{F\}}\text{(if)}$$

$$F = am \leq n < bm \qquad H = G[{}^{\frac{a+b}{2}}_{d}] \qquad G = F \wedge d = \tfrac{a+b}{2}$$

$$\dfrac{\dfrac{\overset{(1)}{F \supset H} \quad \overset{(zw)}{\{H\}\, d := \frac{a+b}{2}\, \{G\}}}{\{F\}\, d := \frac{a+b}{2}\, \{G\}}\text{(imp)} \quad \{G\}\, \texttt{if}\ dm{\leq}n\ \texttt{then}\ \cdots\, \{F\}}{\{F\}\, d := \frac{a+b}{2}; \texttt{if}\ dm \leq n\ \texttt{then}\ a := d\ \texttt{else}\ b := d\, \{F\}}\text{(ha)}$$

Zeige die Gültigkeit von ...

(1) $F \supset G[{}^{\frac{a+b}{2}}_{d}]$

(2) $(G \wedge dm \leq n) \supset F[{}^d_a]$

(3) $(G \wedge dm \not\leq n) \supset F[{}^d_b]$

$\{am \leq n < bm\}$ $d := (a+b)/2;$ `if` $dm \leq n$ `then` $a := d$ `else` $b := d$ $\{am \leq n < bm\}$

$\{\, F : am \leq n < bm \,\}$
$d := (a+b)/2;$
$\{\, F_5 : F \wedge d = (a+b)/2 \,\}$ (zw′ ↓)
`if` $dm \leq n$ `then`
  $\{\, F_6 : F \wedge d = (a+b)/2 \wedge dm \leq n \,\}$ (if↓)
  $\{\, F_3 : F[{}^{d}_{a}] \,\}$ (zw↑)
  $a := d$
  $\{\, F_2 : F \,\}$ (endif↑)
`else`
  $\{\, F_7 : F \wedge d = (a+b)/2 \wedge dm \not\leq n \,\}$ (if↓)
  $\{\, F_4 : F[{}^{d}_{b}] \,\}$ (zw↑)
  $b := d$
  $\{\, F_1 : F \,\}$ (endif↑)
`// end if`
$\{\, F : am \leq n < bm \,\}$

Zeige die Gültigkeit von . . .

$F_6 \supset F_3$: $(F \wedge d = (a+b)/2 \wedge dm \leq n) \supset F[{}^{d}_{a}]$

$F_7 \supset F_4$: $(F \wedge d = (a+b)/2 \wedge dm \not\leq n) \supset F[{}^{d}_{b}]$

Abort-Anweisung – Partielle Korrektheit

„$\{F\}\,\texttt{abort}\,\{G\}$ ist partiell korrekt“ bedeutet:
Für alle Zustände $\sigma \in \mathcal{S}$ gilt:
Wenn $[F]\,\sigma$ wahr und $[\texttt{abort}]\,\sigma$ definiert ist, dann ist $[G]\,[\texttt{abort}]\,\sigma$ wahr.

Da $[\texttt{abort}]\,\sigma$ für alle $\sigma \in \mathcal{S}$ undefiniert ist,
ist die Implikation immer wahr, egal wie F und G aussehen.

$\{F\}\,\texttt{abort}\,\{G\}$ (ab) ist ein zulässiges Axiom für partielle Korrektheit.

$\{1\}\,\texttt{abort}\,\{0\}$ (ab) alternatives Axiom für partielle Korrektheit

$$\frac{F \supset 1 \quad \{1\}\,\texttt{abort}\,\{0\} \quad 0 \supset G}{\{F\}\,\texttt{abort}\,\{G\}}\ (\text{imp})$$

Annotierungsregel

$$\texttt{abort} \quad \mapsto \quad \begin{array}{l} \{1\} \quad \text{ab} \\ \texttt{abort} \\ \{0\} \quad \text{ab} \end{array}$$

Abort-Anweisung – Totale Korrektheit

„$\{F\}\,\texttt{abort}\,\{G\}$ ist total korrekt“ bedeutet:
Für alle Zustände $\sigma \in \mathcal{S}$ gilt:
Wenn $[F]\,\sigma$ wahr ist, dann ist $[\texttt{abort}]\,\sigma$ definiert und $[G]\,[\texttt{abort}]\,\sigma$ wahr.

Da $[\texttt{abort}]\,\sigma$ für alle $\sigma \in \mathcal{S}$ undefiniert ist,
ist die Implikation nur wahr, wenn $[F]\,\sigma$ immer falsch ist.

$\{0\}\,\texttt{abort}\,\{G\}$ (abt) ist ein zulässiges Axiom.

$\{0\}\,\texttt{abort}\,\{0\}$ (ab) alternatives Axiom

$$\frac{0 \supset 0 \qquad \{0\}\,\texttt{abort}\,\{0\} \qquad 0 \supset G}{\{0\}\,\texttt{abort}\,\{G\}}\ \text{(imp)}$$

Annotierungsregel

$$\texttt{abort} \quad \mapsto \quad \begin{array}{ll} \{0\} & \text{abt} \\ \texttt{abort} & \\ \{0\} & \text{abt} \end{array}$$

$\{x > 2\}$ `if` $x \geq 0$ `then` ε `else abort` $\{x > 2\}$ ist total korrekt.

$\{ x > 2 \}$
`if` $x \geq 0$ `then`
  $\{ x > 2 \wedge x \geq 0 \}$ if $\downarrow$
  $\{ x > 2 \}$ leer $\uparrow$
  ε
  $\{ x > 2 \}$ endif $\uparrow$
`else`
  $\{ x > 2 \wedge x \not\geq 0 \}$ if $\downarrow$
  $\{ 0 \}$ abt
  `abort`
  $\{ 0 \}$ abt
  $\{ x > 2 \}$ endif $\uparrow$
`// end if`
$\{ x > 2 \}$

Zu zeigen: Die Formeln

$$x > 2 \wedge x \geq 0 \supset x > 2 \quad \text{(imp)}$$

$$x > 2 \wedge x \not\geq 0 \supset 0 \quad \text{(imp)}$$

$$0 \supset x > 2 \quad \text{(imp)}$$

sind gültig.

$\{x > 2\}$ `if` $x \geq 0$ `then` ε `else abort` $\{x > 2\}$ ist partiell korrekt.

$\{\, x > 2 \,\}$
`if` $x \geq 0$ `then`
  $\{\, x > 2 \wedge x \geq 0 \,\}$ if $\downarrow$
  $\{\, x > 2 \,\}$ leer $\uparrow$
  ε
  $\{\, x > 2 \,\}$ endif $\uparrow$
`else`
  $\{\, x > 2 \wedge x \not\geq 0 \,\}$ if $\downarrow$
  $\{\, 1 \,\}$ ab
  `abort`
  $\{\, 0 \,\}$ ab
  $\{\, x > 2 \,\}$ endif $\uparrow$
`// end if`
$\{\, x > 2 \,\}$

Zu zeigen: Die Formeln

$$\begin{array}{l} \ x > 2 \wedge x \geq 0 \\ \supset x > 2 \end{array} \quad \text{(imp)}$$

$$\begin{array}{l} \ x > 2 \wedge x \not\geq 0 \\ \supset 1 \end{array} \quad \text{(imp)}$$

$$\begin{array}{l} \ 0 \\ \supset x > 2 \end{array} \quad \text{(imp)}$$

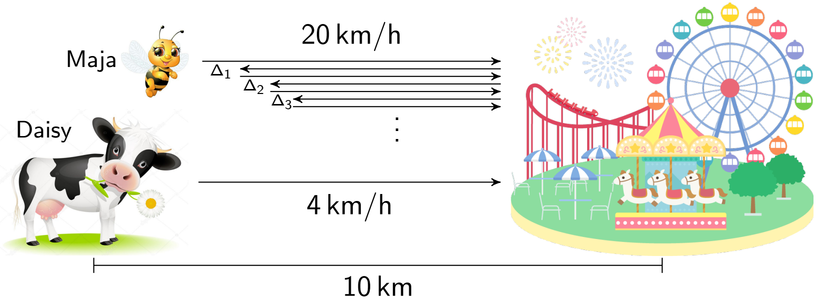
sind gültig.

Partielle Korrektheit von Schleifen

$\{F\}$
$\{e \wedge F_i\}$
$\Pi;$
$\{e \wedge F_{i-1}\}$
$\Pi;$
$\{e \wedge F_{i-2}\}$
$\vdots$
$\{e \wedge F_1\}$
$\Pi;$
$\{\neg e \wedge G =: F_0\}$
$\{G\}$

Sonst: Argumentiere rückwärts.
Beweise $F \supset F_i$ für beliebiges i.
Kaum machbar.

Welchen Weg legt Maja zurück?



Fokus auf das Gemeinsame: die vergangene Zeit

$$Zeit_{\text{Maja}} = Zeit_{\text{Daisy}} \qquad Zeit = \frac{Weg}{Geschwindigkeit}$$

$$\frac{Weg_{\text{Maja}}}{20\,\text{km/h}} = \frac{Weg_{\text{Daisy}}}{4\,\text{km/h}}$$

$$Weg_{\text{Maja}} = Weg_{\text{Daisy}} \cdot \frac{20\,\text{km/h}}{4\,\text{km/h}} = 10\,\text{km} \cdot \frac{20\,\text{km/h}}{4\,\text{km/h}} = 50\,\text{km}$$

Biene: www.pinterest.at/pin/368521181991009023, Hochschaubahn: www.cleanpng.com (contributor Onacy)
Kuh: www.alamy.com (image id BTNTM8)

Partielle Korrektheit von Schleifen

$\{F\}$
$\{e \wedge Inv\}$
$\Pi;$
$\{e \wedge Inv\}$
$\Pi;$
$\{e \wedge Inv\}$
$\vdots$
$\{e \wedge Inv\}$
$\Pi;$
$\{\neg e \wedge Inv\}$
$\{G\}$

Anzahl der Durchläufe irrelevant.
Partielle Korrektheit bewiesen!

Idee: Finde Formel *Inv* sodass

- $\{e \wedge Inv\}$ Π $\{Inv\}$ korrekt
- $F \supset Inv$ gültig
- $\neg e \wedge Inv \supset G$ gültig

Inv . . . "Invariante"

Partielle Korrektheit von Schleifen

$$\frac{\{\mathit{Inv} \land e\}\,\Pi\,\{\mathit{Inv}\}}{\{\mathit{Inv}\}\,\texttt{while}\ e\ \texttt{do}\ \Pi\,\{\mathit{Inv} \land \neg e\}}\ \text{(wh)}$$

Annotierung:

$$\begin{array}{lllll} & \texttt{while}\ e\ \texttt{do} & & \cdots & \\ \mapsto \{\mathit{Inv}\} & \texttt{while}\ e\ \texttt{do} & \{\mathit{Inv}\land e\} & \cdots\ \{\mathit{Inv}\} & \{\mathit{Inv}\land\neg e\} \end{array} \qquad \text{(wh)}$$

Inv: “Schleifeninvariante”; gilt vor, während und nach der Schleife.

Die Schleifenaussage ist korrekt, wenn eine Ableitung mit irgendeiner Formel *Inv* möglich ist.
Das Finden einer geeigneten Invarianten kann schwierig sein.

Multiplikation

$\{ F_1 \colon y = y_0 \}$
$\{ F_9 \colon Inv\left[{0 \atop z}\right] \}$
$z := 0;$
$\{ F_3 \colon Inv \}$
$\texttt{while}\ y \neq 0\ \texttt{do}\ \{$
$\quad \{ F_4 \colon Inv \wedge y \neq 0 \}$
$\quad \{ F_8 \colon Inv\left[{y-1 \atop y}\right]\left[{z+x \atop z}\right] \}$
$\quad z := z + x;$
$\quad \{ F_7 \colon Inv\left[{y-1 \atop y}\right] \}$
$\quad y := y - 1$
$\quad \{ F_5 \colon Inv \}$
$\}$
$\{ F_6 \colon Inv \wedge \neg(y \neq 0) \}$
$\{ F_2 \colon z = x * y_0 \}$

Zu tun:

- Finde geeignete Invariante *Inv*
- Zeige Gültigkeit von
  $F_1 \supset F_9$, $F_4 \supset F_8$, $F_6 \supset F_2$

Wir wählen $Inv = (xy + z = c)$
(für ein geeignetes konstantes c)

$F_1 \supset F_9$: $y = y_0 \supset Inv[^0_z]$

$$y = y_0 \supset (xy + 0 = c)$$

Gültig, wenn wir $c = xy_0$ wählen.

$F_4 \supset F_8$: $(Inv \wedge y \neq 0) \supset Inv[^{y-1}_{y}][^{z+x}_{z}]$

$$(Inv \wedge y \neq 0) \supset (xy + z = xy_0)[^{y-1}_{y}][^{z+x}_{z}]$$

$$(Inv \wedge y \neq 0) \supset (x(y-1) + (z+x) = xy_0)$$

$$(Inv \wedge y \neq 0) \supset (xy + z = xy_0)$$

Gültig.

$F_6 \supset F_2$: $(Inv \wedge \neg(y \neq 0)) \supset z = xy_0$

$$(xy + z = xy_0 \wedge y = 0) \supset z = xy_0$$

$$(0 + z = xy_0 \wedge y = 0) \supset z = xy_0$$

Gültig.

Erster Beweis für die Korrektheit des Programms!
(partielle Korrektheit)

Termination

```
while y ≠ 0 do {
  z := z + x;
  y := y − 1
}
```

Warum terminiert diese Schleife?

- Schleifenbedingung ist falsch, falls $\sigma(y) = 0$.
- Schleifenanweisungen verringern den Wert von y um 1.
- Daher terminiert die Schleife, wenn y zu Beginn nicht negativ ist.

Allgemein:

- Finde Ausdruck für die „Größe" des aktuellen Zustands $\quad |\sigma| = \sigma(y)$
- Zeige, dass sich die Größe in jedem Durchlauf ganzzahlig verringert.
  $\sigma, \sigma' =$ Zustand vor/nach Durchlauf $\quad |\sigma'| < |\sigma|$
- Zeige, dass die Größe nach unten beschränkt ist. $\quad |\sigma| \geq 0$

Dann terminiert die Schleife sicher.

Totale Korrektheit von Schleifen

t: „Terminationsfunktion“ oder „Variante“, gibt die Größe des momentanen Zustands an.

Der Wert des Ausdrucks t muss

- ... eine ganze Zahl sein,
- ... nicht-negativ vor und während der Schleife sein und
- ... in jedem Schleifendurchlauf strikt kleiner werden.

$$\frac{\overset{\text{partielle Korrektheit}}{\{Inv \wedge e\}\,\Pi\,\{Inv\}} \qquad \overset{\text{Variante wird strikt kleiner}}{\{Inv \wedge e \wedge t = t_0\}\,\Pi\,\{t < t_0\}} \qquad \overset{\text{und ist beschränkt}}{Inv \supset t \geq 0}}{\{Inv\}\ \texttt{while}\ e\ \texttt{do}\ \Pi\,\{Inv \wedge \neg e\}}$$

t_0: Hilfsvariable, repräsentiert den Wert von t vor der Iteration

Alternative Formulierungen

$$\frac{\{Inv \land e \land t = t_0\}\,\Pi\,\{Inv \land 0 \leq t < t_0\}}{\{Inv\}\,\texttt{while}\ e\ \texttt{do}\ \Pi\,\{Inv \land \neg e\}}\ \text{(wht)}$$

$$\frac{\{Inv \land e \land t = t_0\}\,\Pi\,\{Inv \land (e \supset 0 \leq t < t_0)\}}{\{Inv\}\,\texttt{while}\ e\ \texttt{do}\ \Pi\,\{Inv \land \neg e\}}\ \text{(wht')}$$

Annotierung:

$$\begin{array}{ll} & \texttt{while}\ e\ \texttt{do}\ \ \Pi \qquad \mapsto \qquad \text{(wht)} \\ \{Inv\} & \texttt{while}\ e\ \texttt{do}\ \{Inv \land e \land t = t_0\}\ \Pi\ \{Inv \land\ \ 0 \leq t < t_0\ \}\ \ \{Inv \land \neg e\} \end{array}$$

$$\begin{array}{ll} & \texttt{while}\ e\ \texttt{do}\ \ \Pi \qquad \mapsto \qquad \text{(wht')} \\ \{Inv\} & \texttt{while}\ e\ \texttt{do}\ \{Inv \land e \land t = t_0\}\ \Pi\ \{Inv \land (e \supset 0 \leq t < t_0)\}\ \ \{Inv \land \neg e\} \end{array}$$

Multiplikation

$\{ F_1 \colon y = y_0 \}$
$\{ F_9 \colon Inv[^0_z] \}$
$z := 0;$
$\{ F_3 \colon Inv \}$
`while` $y \neq 0$ `do {`
$\{ F_4 \colon Inv \wedge y \neq 0 \wedge t = t_0 \}$
$\{ F_8 \colon (Inv \wedge 0 \leq t < t_0)[^{y-1}_y][^{z+x}_z] \}$
$z := z + x;$
$\{ F_7 \colon (Inv \wedge 0 \leq t < t_0)[^{y-1}_y] \}$
$y := y - 1$
$\{ F_5 \colon Inv \wedge 0 \leq t < t_0 \}$
`}`
$\{ F_6 \colon Inv \wedge \neg(y \neq 0) \}$
$\{ F_2 \colon z = x * y_0 \}$

Wir wählen:

- $Inv = (xy + z = xy_0)$
- $t = y$

Zu zeigen: $F_1 \supset F_9$, $F_4 \supset F_8$, $F_6 \supset F_2$

$F_1 \supset F_9$ und $F_6 \supset F_2$: siehe vorhin.

$F_4 \supset F_8$ besteht aus zwei Teilen.

$F_4 \supset F_8^p$ (partielle Korrektheit):
$Inv \wedge y \neq 0 \qquad \supset Inv[\,][\,]$
Siehe vorhin.

$F_4 \supset F_8^t$ (Termination):
$Inv \wedge y \neq 0 \wedge t = t_0 \supset 0 \leq t[\,][\,] < t_0$
Noch nicht bewiesen.

$F_4 \supset F_8^t$: $(Inv \wedge y \neq 0 \wedge t = t_0) \supset 0 \leq t[{}^{y-1}_{y}][{}^{z+x}_{z}] < t_0$

$(Inv \wedge y \neq 0 \wedge y = t_0) \supset 0 \leq y[{}^{y-1}_{y}][{}^{z+x}_{z}] < t_0$

$(Inv \wedge y \neq 0 \wedge y = t_0) \supset 0 \leq y-1 < t_0$

Wir müssen zeigen:

$(Inv \wedge y \neq 0 \wedge y = t_0) \supset y-1 < t_0$ ✓

$(Inv \wedge y \neq 0 \quad) \supset 0 \leq y-1$ nicht gültig!

Wir müssen zwei Formeln verstärken:

- Invariante: $Inv' = (Inv \wedge y \geq 0)$
- Vorbedingung: $F_1' = (y = y_0 \wedge y \geq 0)$

... und alle Implikationen neu beweisen (aber nicht hier und jetzt).

$(Inv \wedge y \geq 0 \wedge y \neq 0) \supset 0 \leq y-1$ Ist nun gültig!

Zweiter Beweis für die Korrektheit des Programms!
(totale Korrektheit)

Ist dieses Programm korrekt?

```
// Multipliziert x und y
z := 0;
while y ≠ 0 do {
  z := z + x;
  y := y − 1
}
```

- ✓ Ist das ein zulässiges Programm? Was alles ist zulässig? ⇒ Syntax
- ✓ Was bedeuten die Programmelemente? ⇒ operationale Semantik
- ✓ Wie ist die Behauptung „Multipliziert x und y" zu interpretieren? ⇒ Korrektheitsaussagen
- ✓ Wie können wir so eine Behauptung (automatisch) beweisen? ⇒ axiomatische Semantik (Hoare-Kalkül)