

Vorlesung

Theoretische Informatik II

Bernhard Beckert

Institut für Informatik



Wintersemester 2007/2008

Dank

Diese Vorlesungsmaterialien basieren zum Teil auf den Folien zu den Vorlesungen von

Katrin Erk (gehalten an der Universität Koblenz-Landau)

Jürgen Dix (gehalten an der TU Clausthal)

Christoph Kreitz (gehalten an der Universität Potsdam)

Ihnen gilt mein herzlicher Dank.

– *Bernhard Beckert, Oktober 2007*

Gödelisierung von Turing-Maschinen

Man kann jeder Turing-Maschine

$$M = (K, \Sigma, \delta, s)$$

eindeutig eine Gödelnummer

$$\langle M \rangle \in \mathbb{N}$$

so zuordnen, dass

- die Kodierungsfunktion
(Berechnung von $\langle M \rangle$ aus den Bestandteilen von M)
- die Dekodierungsfunktionen
(Berechnung der Bestandteile von M aus $\langle M \rangle$)

primitiv rekursiv sind.

Gödelisierung von Turing-Maschinen

Man kann jeder Turing-Maschine

$$M = (K, \Sigma, \delta, s)$$

eindeutig eine Gödelnummer

$$\langle M \rangle \in \mathbb{N}$$

so zuordnen, dass

- die Kodierungsfunktion
(Berechnung von $\langle M \rangle$ aus den Bestandteilen von M)
- die Dekodierungsfunktionen
(Berechnung der Bestandteile von M aus $\langle M \rangle$)

primitiv rekursiv sind.

Gödelisierung der Konfigurationen von Turing-Maschinen

Man kann jeder Konfiguration

$$q, w\underline{a}u$$

einer gegebenen Turing-Maschine M eindeutig eine Gödelnummer

$$\langle C \rangle \in \mathbb{N}$$

so zuordnen, dass

- die Kodierungsfunktion
(Berechnung von $\langle C \rangle$ aus den Bestandteilen von C)
- die Dekodierungsfunktionen
(Berechnung der Bestandteile von C aus $\langle C \rangle$)

primitiv rekursiv sind.

Gödelisierung der Konfigurationen von Turing-Maschinen

Man kann jeder Konfiguration

$$q, w\underline{a}u$$

einer gegebenen Turing-Maschine M eindeutig eine Gödelnummer

$$\langle C \rangle \in \mathbb{N}$$

so zuordnen, dass

- die Kodierungsfunktion
(Berechnung von $\langle C \rangle$ aus den Bestandteilen von C)
- die Dekodierungsfunktionen
(Berechnung der Bestandteile von C aus $\langle C \rangle$)

primitiv rekursiv sind.

Lemma 13.1 (Simulationslemma)

Es gibt eine **primitiv rekursive** Funktion

$$f_U : \mathbb{N}^3 \rightarrow \mathbb{N} ,$$

so dass für jede Turing-Maschine M gilt:

Sind C_0, \dots, C_t ($t \geq 0$) Konfigurationen von M mit

$$C_i \vdash_M C_{i+1} \quad (0 \leq i < t) ,$$

dann gilt

$$f_U(\langle M \rangle, \langle C_0 \rangle, t) = \langle C_t \rangle$$

Lemma 13.1 (Simulationslemma)

Es gibt eine **primitiv rekursive** Funktion

$$f_U : \mathbb{N}^3 \rightarrow \mathbb{N} ,$$

so dass für jede Turing-Maschine M gilt:

Sind C_0, \dots, C_t ($t \geq 0$) Konfigurationen von M mit

$$C_i \vdash_M C_{i+1} \quad (0 \leq i < t) ,$$

dann gilt

$$f_U(\langle M \rangle, \langle C_0 \rangle, t) = \langle C_t \rangle$$

Lemma 13.1 (Simulationslemma)

Es gibt eine **primitiv rekursive** Funktion

$$f_U : \mathbb{N}^3 \rightarrow \mathbb{N} ,$$

so dass für jede Turing-Maschine M gilt:

Sind C_0, \dots, C_t ($t \geq 0$) Konfigurationen von M mit

$$C_i \vdash_M C_{i+1} \quad (0 \leq i < t) ,$$

dann gilt

$$f_U(\langle M \rangle, \langle C_0 \rangle, t) = \langle C_t \rangle$$

Beweisidee

- Die Kodierungs- und Dekodierungsfunktionen für Turing-Maschinen und Konfigurationen sind primitiv rekursiv.
- Ein einzelner Schritt der Turing-Maschine ist primitiv rekursiv
- Eine vorgegebene Anzahl t von Schritten ist primitiv rekursiv

Also ist f_U primitiv rekursiv.

(Detaillierter, konstruktiver Beweis durch explizite Angabe der Funktionen:
4 Seiten in [Erk, Priese])

Beweisidee

- Die Kodierungs- und Dekodierungsfunktionen für Turing-Maschinen und Konfigurationen sind primitiv rekursiv.
- Ein einzelner Schritt der Turing-Maschine ist primitiv rekursiv
- Eine vorgegebene Anzahl t von Schritten ist primitiv rekursiv

Also ist f_U primitiv rekursiv.

(Detaillierter, konstruktiver Beweis durch explizite Angabe der Funktionen:
4 Seiten in [Erk, Priese])

Beweisidee

- Die Kodierungs- und Dekodierungsfunktionen für Turing-Maschinen und Konfigurationen sind primitiv rekursiv.
- Ein einzelner Schritt der Turing-Maschine ist primitiv rekursiv
- Eine vorgegebene Anzahl t von Schritten ist primitiv rekursiv

Also ist f_U primitiv rekursiv.

(Detaillierter, konstruktiver Beweis durch explizite Angabe der Funktionen:
4 Seiten in [Erk, Priese])

Beweisidee

- Die Kodierungs- und Dekodierungsfunktionen für Turing-Maschinen und Konfigurationen sind primitiv rekursiv.
- Ein einzelner Schritt der Turing-Maschine ist primitiv rekursiv
- Eine vorgegebene Anzahl t von Schritten ist primitiv rekursiv

Also ist f_U primitiv rekursiv.

(Detaillierter, konstruktiver Beweis durch explizite Angabe der Funktionen:
4 Seiten in [Erk, Priese])

Beweisidee

- Die Kodierungs- und Dekodierungsfunktionen für Turing-Maschinen und Konfigurationen sind primitiv rekursiv.
- Ein einzelner Schritt der Turing-Maschine ist primitiv rekursiv
- Eine vorgegebene Anzahl t von Schritten ist primitiv rekursiv

Also ist f_U primitiv rekursiv.

(Detaillierter, konstruktiver Beweis durch explizite Angabe der Funktionen:
4 Seiten in [Erk, Priese])

TM-berechenbare Funktionen sind μ -rekursiv

Theorem 13.2

Jede TM-berechenbare Funktion ist μ -rekursiv:

$$\mathbf{TM} \subseteq F_{\mu} \quad \text{und} \quad \mathbf{TM}^{part} \subseteq F_{\mu}^{part}$$

Beweisskizze

Sei

$$f : \mathbb{N}^k \rightarrow \mathbb{N}$$

eine TM-berechenbare Funktion.

Sei M eine TM, die f berechnet.

TM-berechenbare Funktionen sind μ -rekursiv

Theorem 13.2

Jede TM-berechenbare Funktion ist μ -rekursiv:

$$\mathbf{TM} \subseteq F_{\mu} \quad \text{und} \quad \mathbf{TM}^{\text{part}} \subseteq F_{\mu}^{\text{part}}$$

Beweisskizze

Sei

$$f : \mathbb{N}^k \rightarrow \mathbb{N}$$

eine TM-berechenbare Funktion.

Sei M eine TM, die f berechnet.

TM-berechenbare Funktionen sind μ -rekursiv

Theorem 13.2

Jede TM-berechenbare Funktion ist μ -rekursiv:

$$TM \subseteq F_\mu \quad \text{und} \quad TM^{part} \subseteq F_\mu^{part}$$

Beweisskizze

Sei

$$f : \mathbb{N}^k \rightarrow \mathbb{N}$$

eine TM-berechenbare Funktion.

Sei M eine TM, die f berechnet.

TM-berechenbare Funktionen sind μ -rekursiv

Beweisskizze

Dann

$$f(n_1, \dots, n_k) = (f_U(\langle M \rangle, \text{start}, \mu_i((f_U(\langle M \rangle, \text{start}, i))_{\text{Zust}} = \langle h \rangle)))_w$$

wobei

$$\text{start} = \langle s, \# \underbrace{|\dots|}_{n_1} \# \dots \# \underbrace{|\dots|}_{n_k} \# \rangle$$

$\langle h \rangle$ = Gödelisierung des Haltezustands

$(\cdot)_{\text{Zust}}$ = Dekodierung des Zustands in einer Konfiguration

$(\cdot)_w$ = Dekodierung des Wortes links vom Schreib-/Lesekopf

TM-berechenbare Funktionen sind μ -rekursiv

Beweisskizze

Dann

$$f(n_1, \dots, n_k) = (f_U(\langle M \rangle, \text{start}, \mu_i((f_U(\langle M \rangle, \text{start}, i))_{\text{Zust}} = \langle h \rangle)))_w$$

wobei

$$\text{start} = \langle s, \# \underbrace{|\dots|}_{n_1} \# \dots \# \underbrace{|\dots|}_{n_k} \# \rangle$$

$\langle h \rangle$ = Gödelisierung des Haltezustands

$(\cdot)_{\text{Zust}}$ = Dekodierung des Zustands in einer Konfiguration

$(\cdot)_w$ = Dekodierung des Wortes links vom Schreib-/Lesekopf

Korollar (Kleenesche Normalform)

Für jede μ -rekursive Funktion f gibt es primitiv rekursive Funktionen g, h , so dass

$$f(\mathbf{n}) = g(\mu i(h(\mathbf{n}) = 0))$$

also

$$f = g \circ \mu h$$

Teil III

- 1 Einführung
- 2 Primitiv rekursive Funktionen
- 3 $\wp = \text{LOOP}$
- 4 μ -rekursive Funktionen
- 5 $F_\mu = \text{TM} = \text{WHILE}$
- 6 Zusammenfassung

Teil III

- 1 Einführung
- 2 Primitiv rekursive Funktionen
- 3 $\wp = \text{LOOP}$
- 4 μ -rekursive Funktionen
- 5 $F_\mu = \text{TM} = \text{WHILE}$
- 6 Zusammenfassung**

Klassen berechenbarer Funktionen

- **LOOP** = $\emptyset \subsetneq$ **WHILE** = **GOTO** = **TM** = F_μ
- **WHILE**^{part} = **GOTO**^{part} = **TM**^{part} = F_μ^{part}

Klassen berechenbarer Funktionen

- **LOOP** = $\emptyset \subsetneq$ **WHILE** = **GOTO** = **TM** = F_μ
- **WHILE**^{part} = **GOTO**^{part} = **TM**^{part} = F_μ^{part}

Teil IV

Der λ -Kalkül

Zweck

- **Formale Definition von Funktionen**
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

Zweck

- Formale Definition von Funktionen
- **Untersuchung und Anwendung von Funktionsdefinitionen**
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

Die Idee

Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- **Berechnungsmodell**
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

Die Idee

Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- **Grundlage funktionaler Programmiersprachen**
- Einfach und doch mächtig

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren



Die Idee

Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- **Einfach und doch mächtig**

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren



Die Idee

Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren



Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- **Funktionen angewendet auf Funktionen**
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- **Nichts außer Funktionen (keine anderen Datentypen)**

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

Zweck

- Formale Definition von Funktionen
- Untersuchung und Anwendung von Funktionsdefinitionen
- Berechnungsmodell
- Grundlage funktionaler Programmiersprachen
- Einfach und doch mächtig

Grundlegende Eigenschaften

- Funktion identifiziert mit λ -Ausdrücken, die sie beschreiben
- Funktionen angewendet auf Funktionen
- Nichts außer Funktionen (keine anderen Datentypen)

Entwickelt von Alonzo Church und Stephen Kleene in den 1930er Jahren

Definition 14.1 (λ -Ausdruck)

Variable Jede Variable

x

ist ein λ -Ausdruck

Abstraktion Ist x eine Variable und e ein λ -Ausdruck, dann ist

$\lambda x. e$

ein λ -Ausdruck

Anwendung Sind e_1, e_2 λ -Ausdrücke, dann ist

$e_1 e_2$

ein λ -Ausdruck

Definition 14.1 (λ -Ausdruck)

Variable Jede Variable

x

ist ein λ -Ausdruck

Abstraktion Ist x eine Variable und e ein λ -Ausdruck, dann ist

$\lambda x.e$

ein λ -Ausdruck

Anwendung Sind e_1, e_2 λ -Ausdrücke, dann ist

$e_1 e_2$

ein λ -Ausdruck

Definition 14.1 (λ -Ausdruck)

Variable Jede Variable

$$x$$

ist ein λ -Ausdruck

Abstraktion Ist x eine Variable und e ein λ -Ausdruck, dann ist

$$\lambda x. e$$

ein λ -Ausdruck

Anwendung Sind e_1, e_2 λ -Ausdrücke, dann ist

$$e_1 e_2$$

ein λ -Ausdruck

Assoziativität und Bindungsstärke

- Anwendungen sind linksassoziativ:

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

- Anwendung bindet stärker als Abstraktion:

$$\lambda x. e_1 e_2 = \lambda x. (e_1 e_2)$$

WICHTIG!

Dies muss man wissen, um λ -Ausdrücke verstehen zu können!

Assoziativität und Bindungsstärke

- Anwendungen sind linksassoziativ:

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

- Anwendung bindet stärker als Abstraktion:

$$\lambda x. e_1 e_2 = \lambda x. (e_1 e_2)$$

WICHTIG!

Dies muss man wissen, um λ -Ausdrücke verstehen zu können!

Assoziativität und Bindungsstärke

- Anwendungen sind linksassoziativ:

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

- Anwendung bindet stärker als Abstraktion:

$$\lambda x. e_1 e_2 = \lambda x. (e_1 e_2)$$

WICHTIG!

Dies muss man wissen, um λ -Ausdrücke verstehen zu können!

Assoziativität und Bindungsstärke

- Anwendungen sind linksassoziativ:

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

- Anwendung bindet stärker als Abstraktion:

$$\lambda x. e_1 e_2 = \lambda x. (e_1 e_2)$$

WICHTIG!

Dies muss man wissen, um λ -Ausdrücke verstehen zu können!