Theoretische Grundlagen von Programmiersprachen WS-2010/11

Mitschrift von Raphael Bruns

2. Februar 2011

Inhaltsverzeichnis

0 Einführung

Syntax	Semantik
λ-Kalkül	Lisp
kombinatorische Logik	FT-Systeme(Bakkus)
Horn-Klauseln	Prolog

Übungen von 10-12 und 12-14, SR 7 (Raum wird gegebenfalls noch geändert)

1 Definition von Programmiersprachen

Syntax	Semantik
Beschreibung des formalen Aufbaus eines Programms,erlaubte Zeichen, Schlüsselworte	Bedeutung des Programms z.B.: E/A -Funktion

1.1 Beschreibung der Syntax

Eine kontextfreie Grammatik G ist gegeben durch G = (N,T,P,S)
mit $N \neq \emptyset$ endliche Menge von nichtterminalen Zeichen (Variablen, Hilfszeichen, dürfen vom Programmiere nicht benutzt werden.)
T : $N \cap T = \emptyset$ endliche Menge von Terminalzeichen. (Zeichen eines Programms)
(*) $P \subseteq N \times (T \cup N)^*$ endliche Menge von Produktionen (Regeln)
$S \in N$ Axiom (Startsymbol)
$(n, \alpha) \in P \sim n \to \alpha \sim n ::= \alpha, n \in N, \alpha \in (N \cup T)^*, N \cup T \sim A^*$
(A^* ist die Menge aller Zeichenreihen (Worte) über A ($\epsilon \in A^*$) leere Wort)
$A^+ \sim A^*$ ohne ϵ
kontextfreie Grammatik $\sim$ Chomsky-2-Grammatik

Definition U heißt auf v unmittelbar reduzierbar ($u \leftarrow v$), wenn gilt
u = $w\alpha z$
v = w A z mit w,z,αA^*, (A,α) $\in$ P

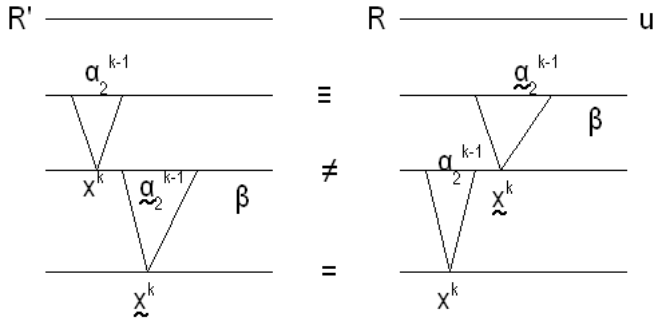
U:	w	α	z
=			
V:	w	A	z

Reduzieren: ausgehend von u wird v erzeugt (durch Anwendung einer Produktion)
Ableiten: ausgehend von v wird u erzeugt (durch Anwendung einer Produktion)
Entsprechende Definition für unmittelbar ableitbar.
$\overset{+}{\leftarrow}$ transitive Hülle von $\leftarrow$, $u = u_1 \leftarrow u_2 \leftarrow .. \leftarrow u_n = v$ (mindestens ein Übergang)
$\overset{*}{\leftarrow}$ transitive reflexive Hülle (entweder kein oder mindestens 1 Übergang)

Definition Die von G erzeugte Sprache L(G) ist gegeben durch $L(X) := \{u | u \in T^* \text{ und } u \overset{+}{\leftarrow} S'\}$
$u \in L(G)$ heißen Sätze (syntaktisch korrekte Programme) $u \in A^*$ mit $u \overset{*}{\leftarrow}$ heißen Satzformen.
Zwei Grammatiken heißen äquivalent, wenn sie die gleiche Sprache erzeugen.

Defintion Eine Reduktionschritt (Ableitungsschritt) ist ein Tripel $(w, A ::= \alpha; z)$ mit $w, z \in A^*$ und $A ::= \alpha \in P$
Eine Reduktionsfolge (Ableitungsfolge) von u nach v ist eine nichtleere Folge $(w_1 A ::= \alpha_1, z_1)$,..,$(w_n A_n ::= \alpha_n, z_n)$, $n \geq 1$, (n $\sim$ Länge der Reduktionsfolge) mit $u = u_0 = w_1 \alpha_1 z_1$, $v = u_n = w_n A_n z_n$ und $u_i = w_i A_i z_i = w_{i+1} \alpha_{i+1} z_{i+1}$, $i = 1, .., n-1$

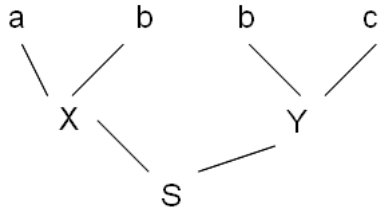
Definition R und R' seinen Reduktionsfolgen von u nach v der Länge n.
$R \lessdot R'$ heißt unmittelbar Kaninischer <> wenn R und R' identisch sind, bist auf den K,K+1-ten Reduktionsschritt, mit der Bedingung R' R.
Unterschied: Reihenfolge der Anwendung 2er unterschiedlicher Regeln.



kanonisch: „linkeste“ Möglichkeit zu erst.

Definition R,R' heißen unwesentlich verschieden, wenn eine Serie von Reduktionsschritten existiert $R = R_1, R_2, .., R_m = R'$ mit $R_i \lessdot R_{i+1}$ oder $R_{i+1} < R_i$.

Strukturbaum: $G = (\{S, X, Y\}, \{a, b, c\}, P, S), P : S ::= XY | aY, Y ::= bY | bc, X ::= ab$



Satz Zwei Reduktionsfolgen von einem Satz $u \in L(G)$ zu einem Axiom S' sind genau dann unwesentlich verschieden, wenn ihre Strukturbäume identisch sind.

Definition Ein Satz $u \in L(G)$ heißt eindeutig, genau dann, wenn alle Reihenfolgen von u nach s unwesentlich verschieden sind!. Eine Grammatik G heißt eindeutig, wenn alle Programme aus L(G) eindeutig sind.
Unabdingbar bei der Konstruktion von Programmiersprachen!!!

```
if <boolean expression> then for ... do
     if < boolean expression> then <unconditional statement>
     else <statement>
```

Produktionen

```
<if clause> ::= if < boolean expresssion> then
<unconditional statement> ::= <for statement>
<if statement> ::= <if clause> <undonditional statement>
<statement> ::= <conditional statement>
<conditional statement> ::= <if statement> | <if statement> else <statement>
```

für $< be > \approx 7 < 5$

a) Rest ist Dummy

b) statement hinter dem letzten else wird immer ausgeführt.

Korrektur 1963:

```
<cs> ::= <fs> wurde ersetzt durch
<cs> ::= <if clause> <for statement>
```

$\Rightarrow$ a wurde für korrekt erklärt
Regeln: $A ::= \alpha, \alpha \in (N \cup T)^*$

2 Vereinfachung von kontextfreien Sprachen

2.1 Entfernung von nutzlosen Symbolen

Definition: Ein Symbol X heißt nützlich, wenn es eine Ableitung $S \xrightarrow{*} \alpha X \beta \xrightarrow{*} w$, $w \in T$ gibt. Andernfalls ist X nutzlos.

2 Bedinungen:

HS1: Zu einer gegebenen kontextfreien Grammatik G, $G = (N, T, P, S)$, $L(G) \neq \emptyset$, gibt es eine äquivalente kontextfreie Grammatik G, $G' = (N', T, P', S)$, so daß es für jedes $A \in N'$ ein $w \in T^*$ mit $A \xrightarrow{x} w$.
Bestimme alle A aus N'
begin
AltN:= $\emptyset$
NeuN:= $\{A | A \to w$ für ein $w \in T^*\}$;
while AltN $\neq$ NeuN do
begin
AltN:= NeuN
NeuN:= AltN $\cup \{A | A \to \alpha$ für ein $\alpha \in T \cup AltN\}^*$

end
N':= NeuN
end
Bild1

HS2: Zu einer gegebenen $G = (N, T, P, S)$ läßt sich Konstruktiv eine äquivalente Grammatik $G = (N', T', P', S)$ angeben, so daß für jedes $X \in N' \cup T'$ ein α und ein $\beta \in (N' \cup T')^*$ existiert mit $S \overset{*}{\leftarrow} \alpha X \beta$.

Beweisskizze: Konstruktion von N' und T'

1. $S \in N'$
2. $A \in N'$ und $A ::= \alpha_1 | \alpha_2 | .. | \alpha_n$
   $\Rightarrow$ Alle Nichtterminale der α_i sind Elemente von N'
   Alle Terminale der α_i sind Elemente von T'
   P' ist die Menge der Regeln von P, die nur Symbole von $N' \cup T'$ enthalten.

Satz: Zu jeder kontextfreien Grammatik G läßt sich effektiv eine kontextfreie Grammatik G' ohne nutzlose Symbole erzeugen.

Beweisskizze: Zunächst Anwendung von HS1, danach Anwendung von HS2.
Reihenfolge ist wichtig!!!

Beispiel: $S :: AB/a, A ::= a$
HS1:B ist auf keine Terminalzeichenreihe ableitbar.
$\Rightarrow$ Entfernung von B + Entfernung der Regel $S := AB$
$\Rightarrow$ es verbleiben $S ::= a, A ::= a$
HS2: Nur S und a kommen in Satzformen vor $\Rightarrow G' = (\{S\}, \{a\}, \{s ::= a\}, S)$

Umgekehrte Reihenfolge: HS2: Alle Symbole kommen in Satzform vor. Ursprüngliche Grammatik bleibt erhalten.
HS1: $S ::= A, A ::= a$, nicht so weit gekürzt, wie in der ersten Variante
Elemination von Epsilonregeln: $A ::= c$, Voraussetzung $\epsilon \notin L(G)$

Satz: Wenn $L = L(G)$ für jede kontextfreie Grammatik $G = (N, T, P, S)$, dann ist $L\epsilon$ eine Sprache $L(G')$ für eine kontextfreien Grammatik G' ohne nutzlose Symbole und ohne $\epsilon-$Produktionen.

1.Schritt Bestimmung der nullierbaren Symbole A ($A \overset{*}{\rightarrow} \epsilon$)
Iterativ: A nullierbar <> $A \overset{*}{\rightarrow} \epsilon$

a) $A ::= \epsilon \in P \Rightarrow A$ ist nullierbar

b) $B ::= \alpha_1 ... \alpha_n \in P$ und alle Symbole α_i sind nullierbar $\Rightarrow B$ ist nullierbar

Bild2

2.Schritt Konstruktion von P'
A::= $X_1 X_2 .. X_n \in P$, $X_i \in N \cup T$
$\Rightarrow$ Ersetze diese Produktionen durch die folgende Menge von Regeln ($A ::= \alpha_1 ... \alpha_n$):

a) X_i nicht nullierbar $\Rightarrow \alpha_i = X_i$

b) X_i ist nullierbar $\Rightarrow$ setze $\alpha_i = X$ und $\alpha_i = \epsilon$

c) nicht alle α_i dürfen ϵ sein.

Beispiel: $A ::= X_1X_2X_3 \in P$ und X_2X_3 sind nullierbar.
$\Rightarrow$ eine neue Menge von Regeln: $A ::= X_1X_2X_3$, $A ::= X_1X_2$, $A ::= X_1X_3$, $A ::= X_1$;
Ergänzung $\epsilon \in L(G)$
zusätzlich $S ::= \epsilon$

2.1.1 Normalformen:

Satz:

a) Jede kontextfreie Grammatik ohne $\epsilon \in L(G)$ wird von einer Grammatik erzeugt, die nur Regeln der Art $A ::= AB$ und $A ::= a$ besitzt mit $A, B, C \in N$, $a \in T$. (Chomsky-Normalform)

b) Jede kontextfreie Grammatik ohne $\epsilon \in L(G)$ wird von einer Grammatik erzeugt, die nur Regeln der Art $A := a\alpha$, $A \in N$, $a \in T$, $\alpha \in N^*$ besitzt. Greibach-Normalform.

2.2 Erkennung einer kontextfreien Sprache

Hilfsmittel: erkennende Automat
1-Weg-Nichtdeterministische Kellerautomat(1-N-KA)
Bild
In Abhängigkeit vom:

- Zustand er Kontrolleinheit
- gelesenes Symbol auf der Eingabe
- obesters Kellersymbol

Kann der 1-N-KA

- Zustand der Kontrolleinheit ändern
- auf der Eingabe ein Symbol nach oben gehen
- oberstes Kellersymbol ersetzen

Definition: Ein 1-N-KA ist ein Tupel (Q,Σ,Γ,δ,q_0,Z_0,F) mit

- Q endliche Menge von Zuständen (Kontrolleinheit)
- Σ endliche Menge von Eingabesymbolen
- Γ endliche Menge von Kellersymbolen
- $q \in Q$ Anfangszustand
- $Z_0 \in \Gamma$ Anfangssymbol Keller
- $F \subseteq Q$ Menge von Endzuständen(kann auch leer sein)

δ: Übergangsfunktion δ: $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to K \subseteq Q \times \Gamma^*$
Σ: Kleinbuchstaben vom Anfang des Alphabets
Elemente aus Σ^* sind Kleinbuchstaben vom Ende des Alphabets.
Γ: Großbuchstaben
Elemente von Γ^* sind griechische Buchstaben
Interpretation eines Übergangs: (Anwendung einer Regel)
$\delta(q, a, Z) = (p, \gamma)$, $q, p \in Q$, $Z \in \Gamma$, $\gamma = \gamma_1\gamma_2..\gamma_n \in \Gamma^*$
$a \in \Sigma$
M mit Zustand q, gelesener Eingabe a und oberstem Kellersymbol Z geht über in den Zustand p, ersetzt Z durch γ (γ_n kommt als erstes in den Keller und γ_1 als letztes) und verschiebt des Lesekopf auf der Eingabe um ein Symbol.
$a = \epsilon$:
Analog: Aber Regel anwendbar unabhängig von gelesenem Symbol auf der Eingabe und Lesekopf bleibt stehen.

Zustandsbeschreibung: Bild
Beschreiben durch: (q,w,γ) mit $\gamma = \gamma_n..\gamma_1$, $w = a_k...a_m$
d.h. es gilt:
(q,$a_k...a_n$,$\gamma_n...\gamma_1$) $\to$ (p,$a_{k+1}...a_m$,$\beta\gamma_n...\gamma_1$)
Falls in δ eine Regel existiert
(q,a_k,γ_n) $\underset{(\to)}{::=}$ (p,β), $\beta \in \Gamma^*$
$\xrightarrow{*}$ transitive reflexive Hülle von $\to$
$\xrightarrow{+}$ transitive Hülle
Startkonfiguration (1-NA-KA)
Bild
1-N-KA Akzeptierte Sprache (2 Varianten) Bild KA

1. Für einen 1-N-KA M ist L(M) die durch einen Endzustand akzeptierte Sprache gegeben durch $L(M) = \{w|(q_0, w, Z_0) \xrightarrow{*} (p, \epsilon, \gamma)$ mit $p \in F, \gamma \in \Gamma^*\}$.
2. Für einen 1-N-KA M ist N(M) die durch den leeren Keller akzeptierte Sprache. $N(M) = \{w|(q_0, w, Z_0) \xrightarrow{*} (p, \epsilon, \epsilon), p \in Q\}$

Satz:

1. Ist $L = L(M_2)$ für einen 1-N-KA, dann ist auch $L = N(M_1)$ für einen 1-N-KA.
2. Ist $L = N(M_1)$ für einen 1-N-KA, dann ist $L = L(M_2)$ für einen 1-N-KA.

Konstruktion eines erkennenden Automaten für eine gegebene kontextfreie Grammatik (Sprache)

Satz: Sei L eine kontextfreie Sprache, dann läßt sich effektiv ein 1-N-KA M konstruieren, mit $L = N(M)$.
Annahme: $\epsilon \notin L(G)$ und $G = (N, T, P, S) \Rightarrow M = (\{q\}, T, N \cup T, \delta, q, S, \emptyset)$
Bild q0 KA

Iteratives Vorgehen:

1. Oberstes Kellersymbol ist Element $\overline{N}$ aus N.

a) Wähle beliebige Regel aus P mit $\overline{N} ::= \alpha_1 ... \alpha_m$

b) Ersetze $\overline{N}$ im Keller durch $\alpha_1 .. \alpha_m$ (α_1 oberstes Symbol)

c) keine Bewegung auf der Eingabe (ϵ-Regel)

2. Oberstes Kellersymbol ist Element $\overline{T}$ aus T
   $\Rightarrow$ Vergleiche $\overline{T}$ mit gelesenem Symbol auf der Eingabe

   a) Beide identisch $\Rightarrow \overline{T}$ löschen und auf der Eingabe ein Symbol weitergehen.

   b) Beide verschieden $\Rightarrow$ Automat bleibt stehen

Formal:

a) $\overline{N} ::= \alpha_1 .. \alpha_m \in P \Rightarrow \delta(q, \epsilon, \overline{N}) = (q, \alpha_1 ... \alpha_m)$

b) $\delta(q, a, a) = (q, \epsilon), a \in T$

Welche Kellerautomaten können welche Sprachen erkennen:

a) $\{a^n b^n\}$ - 1-D-KA

b) $\{a^n b^{2n}\}$ - 1-D-KA

c) $\{a^n b^n\} \cup \{a^n b^{2n}\}$ - nicht von 1-D-KA, aber vom 1-N-KA

d) $\{a^n b^n c^n\}$ - 2-D-KA

2.3 Sematikbeschreibung von Programmiersprachen

1. Die Semantikbeschreibung von statischen Programmteilen ist einfach.

2. Probleme gibt es bei dynamischen Teilen.
   $\Rightarrow$ rekursive Unterprogramme

Prinzip:

1. Annahme die Semantik eines Programms ohne Unterprogramme ist bekannt.

2. Die Semantik (Bedeutung) eines Programms mit Unterprogrammen wird auf die Semantik eines („äquivalenten“) Programmes ohne Unterprogramme zurückgeführt.

Notation für Unterprogramme (Methoden) Eine Methodendeklaration muss umfassen

1. Kennzeichnung der Deklaration durch ein Schlüsselwort (methode)

2. Name

3. Angabe von Parametern

4. Angabe des Programmtextes
   method <Name> (< Liste_von_formalen_Parametern>); begin
   <UP-block>
   end

   Methode-P
   ...

method $p(x_1, .., x_n)$; begin ..... end; ⇒deklarierendes Vorkommen von P
...
Aufruf:
$p(a_1, .., a_n)$; Aufruf angewandtes Vorkommen von P

Methodenrumpf ist ein Block
⇒ Innerhalb des Rumpfes können wieder Methoden deklariert werden.

1. method $p(x_1, .., x_n)$ begin ... x_i ... end; x_i formaler Parameter (Identifikator)
2. a) method $p(x_1, .., x_n)$ begin var a: int; .. a...end;
   b) method $p(x_1, .., x_n)$ begin method $f(y_1, ..., y_m)$; begin...end; ... f(..) end; ⇒ f lokaler Identifikator
3. a) method $p(x_1, .., x_n)$; begin var a:int; method $f(y_1, .., y_m)$ begin a end; .... end; a ist globaler, bzw freie Identifikator in f
   b) method $p(x_1, .., x_n)$; begin method $f(y_1, .., y_n)$ begin....end method $g(z_1, .., z_l)$ begin ... f(..) ... end; ... end ⇒ f ist freie Identifikator in g
4. method $p(x_1, .., x_n)$; begin method $g(y_1, .., y_n)$ begin ... x_i end; end ⇒ x_i ist ein formal globaler Parameter

Gültigkeitsbereiche(„scops“)

1. lokale Identifikatoren: Methodenrumpf
2. formalen Parameter: Methodenrumpf
3. Methodennamen: der die Methodendeklaration unmittelbar umfassenden Block.

Methodenaufrufe:

1. gewöhnlicher Methodenaufruf: <Methodenname> $(a_1, .., a_n)$
2. formaler Methodenaufruf: <Parametername> $(a_1, .., a_n)$
   ...method $p(x_1, .., x_2)$ begin method $f(y_1, .., y_n)$ begin method $g(z_1, ..., z_n)$ begin $y(z_1, .., z_n)$ end method $h(u_1, .., h_2)$ begin .. end.. $f(h, a_2, .., a_n)$ end

Definition:

1. Ein Programm heißt syntaktisches Programm, wenn es auf das Axiom der Grammatik reduzierbar ist.
2. Ein syntaktisches Programm heißt gebunden (oder formal) wenn für jedes angewandete Auftreten eines Identifikators ein entsprechendes (eindeutiges) deklarierendes Auftreten gibt.
3. Das Hauptprogramm $\tilde{\Pi}$ eines gebundenen Programms Π ist der Teil von Π, der nach dem Streichen sämtlicher Methodendeklarationen übrigbleibt.
4. Ein gebundenes Programm heißt übersetzbar (oder zulässig), wenn jedes Auftreten eines Identifikators gemäß der Deklaration erfolgt.
5. Ein übersetztes Programm heißt ausgezeichnet, wenn alle Deklarationen verschiedene Namen besitzen.

2.4 Kopierregelsemantik

Kopierregel: Sei Π ein partiell übersetzbares Programm. Sei $p(a_1, .., a_n)$ ein Aufruf der Methode p, der nicht innerhalb einer Methodendeklaration auftritt. Sei ferner method $p(x_1, .., x_n)$ begin ... σ ... end; die zugehörige Methodendeklaration.
Π: ... method $p(x_1, .., x_n)$; begin σ... end; .... ; $p(a_1, .., a_n)$;...
$\top$[1]
Π' : ... method $p(x_1, .., x_n)$; begin σ end; .... ;begin σ' end;...
Das funktional äquivalente Programm Π' entsteht aus Π durch Anwendung der Kopierregel ($\Pi \vdash\!\!\!\!-\!\!\!\!-_{*} \Pi'$) wie folgt:
Der Aufruf $p(a_1, .., a_n)$ wird durch den modifizierten Rumpf begin σ' end, der aufgerufenen Methode p ersetzt.
Die Modifikationen sind:

1. Substitution: Ersetzen aller formaler Parameter x_i, die in σ auftreten, durch den aktuellen Parameter a_i.
2. Umbenennung: Alle Identifikatoren, die in σ selbst (lokal) gebunden sind, werden unter Verwendung unbenutzter Namen umbenannt.

Die Zeichenreihe begin σ' end heißt erzeugter Block. $IGB(\Pi')$ ist die Menge aller innersten erzeugten Blöcke.

1. Rein syntaktische Transformation auf Zeichen.
2. Umbenennung ist wichtig, um Namenskonflikte zu vermeiden.
3. Die Kopierregel mit Umbenennung beschreibt static scoping (statische Parameterbindung).
4. Gilt $\Pi \dashv \Pi'$ und ist Π übersetzbar, dann ist Π' nicht unbedingt auch übersetzbar.
5. Diese Kopierregel realisiert „call by name“ (Namensparameter).
6. Kopierregeln ohne Umbenennung nennt man naiv.

Π: begin var a: int; method p(x); begin var a: int; a:= 2; x:= x+a; drucke(x) end; a:= 1; p(a); end; $\top$
Π' : begin.... a:= 1; begin var a': int; a' := 2; a:= a + a', end; end; $\Rightarrow$ 3

Naive Kopierregel: Π': ...
a:= 1;
begin var a: int; a:= 2; a:= a + a; drucke(a) end; end; $\Rightarrow$ 4

Π: begin var a: int; method p(x,y); begin y(x); end; p(a,p);
end;
$\top$;
Π': begin ...
begin p(a) end; end; $\Rightarrow$ Fehler wegen (4)

Wird Z umbenannt zu Z', so heißt Z' Kopie von Z.
Die Ausführung eines Programms E_Π eines Programms Π ist gegeben durch $E_\Pi = \{\Pi' | \Pi \vdash\!\!\!\!-\!\!\!\!-_{*} \Pi'\}$

[1] p wird eingesetzt und angepasst

Satz: Gegeben seien Π, Π', Π'' mit $P \overset{}{\vdash\!\!\!\!-\!\!\!\!-}_{\Pi}{}' $ oder Π''
Dann sind entweder Π' und Π'' identisch, oder es existiert ein Programm Π''' mit $\Pi' \dashv \Pi'''$ und $\Pi'' \dashv \Pi'''$
$(E_\Pi, \vdash\!\!\!\!-\!\!\!\!-_*)$ ist ein Verband.
$T_\Pi := \{\Pi' | \Pi \vdash\!\!\!\!-\!\!\!\!-_* \Pi',$ und Π'besitzt höchstens einen innersten erzeugten Block$\}$ heißt Ausführungsbaum von Π.

1. Entfernt man aus einem Programm $\Pi' \in T_\Pi$ (bzw. E_Π) alle Methodendeklarationen, so erhält man das Hauptprogramm Π'_m.

2. Ersetzt man alle Methodenaufrufe in Π'_m durch die unendliche Schleife begin M: goto M end; So erhält man das reduzierte Hauptprogramm Π'_r von Π'_m.

Semantik ist spezifiziert durch Ein/Ausgabefunktion für Programme ohne Methoden. $E/A_\Pi : I \to A$
E/A ist nur partiell definiert, da keine Ausgabe für Eingabewerte, die zu einer Endlosschleife führen.
$\Pi \dashv \Pi' \approx \Pi_r \dashv \Pi'_r$
.. p(..) .. begin σ end M: goto M; .....1. keine neue Endlosschleife, oder 2. mindestens eine neue Endlosschleife.

Definiertheit der E/A-Funktion $\Rightarrow E/A_\Pi \leq E/A_{\Pi'_r}$
$\Rightarrow E/A_\Pi := \bigcup_{\Pi'_r \in E_\Pi} E/A_{\Pi'_r}$ wegen Verbandseigenschaft ist E/A_Π wohldefiniert.

Kopierregel für „call by value“ ... Modifikation des Rumpfes :

1. Substitution:
   a) für jeden Parameter x_i wird unter Verwendung eines neuen Namens eine lokale Hilfsvariable x'_i eingeführt, der zu Begin von σ der aktuelle Ausdruck a_i zugewiesen wird.
   b) Jeders Auftreten von x_i in σ wird durch x'_i ersetzt.

2. Umbenennung
   begin: var a,b : int; method p(x,y); begin x:= (y-x) * y; drucke (x) end; a:= 5; b:= 3; p(b,a-1);
   $\top$
   Π': begin ...
   a::= 5; b:= 3; begin: var x',y': int; Hilfsvariablen.
   x':= b;
   y':= a-1;
   x':= (y'-x')*y'; drucke (x')
   end;

Kopierregel für „call by reference“ ... Modifikationen des Rumpfes

1. Substitution:
   jeder Aktuelle Parameter a_i wird so lange ausgewertet, bis man eine Variable $var(a_i)$ (Adresse) erhält. Danach wird jedes x_i in σ durch die Adresse von a_i ersetzt.

2. Umbenennung
   Π: begin var i: int; M: $array[1..2]$ of int; method p(a,b); begin i:= i+1; a:= a+2; end;
   $M[1] := 10$; $M[2] := 20$;
   i:= 1;
   $p(M[1], M[2])$;
   $Drucke(M[1], M[2])$

   call by name: begin $i := i + 1; M[i] := M[i] + 2$; end;
   $M[2] = 22; M[1] = 10$;

   call by value begin var a',b": int; $a' := M[i]$; $b'' := M[2]$; i:= i+1; a':= a' +2; end;
   $M[2] = 20; M[1] := 10$;

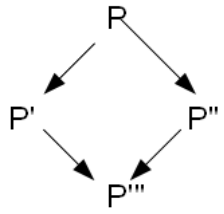
   call by reference begin i:= i+1; $M[1] := M[1] + 2$; end;
   $M[2] = 20$; $M[1] = 12$;

3 λ-Kalkül

Church-Turing-These (ab 1932)
Syntax:Definition einer Funktion: $\lambda xM \approx f(x) = M$
Applikation: $(MN) \approx M(N)$
+ Konstanten und Variablen (Term)

3.1 Einschränkung:

Nur einstellige Funktionen!
Keine Einschränkung!!, da jede mehrstellige Funktion kann durch einstellige Funktionen, bei gewissen Randbedingungen, simuliert werden. (Curryfizieren)
Randbedingung: Existenz von $\underbrace{\text{funktionalen Argumenten und Ergebnissen}}_{\text{Funktional}}$.
$f(x, y) = x - y = -(x, y) \equiv \lambda x(\lambda y(x - y))$: Funktionsdefinition
$((\lambda x(\lambda y(x - y))a)b)$
Einsparung:
$\lambda xy.(x - y)$
Substitution (Operation auf Zeichenreihen) $Sub_N^X[M]$
fordert statische Variablenbindung (Umbenennung)
Gleichheitsbegriff (Konvertierbar)
Semantik (Bedeutung) ist das Ergebnis der Auswertung.
Kopierregel entspricht Reduktionsregel
$P \to R'$ (reduziert) $\approx$ $P \vdash P'$ falls P' aus P entsteht. durch Anwendung einer der beiden folgenden Schritte auf P oder einem Teilterm.
α-Reduktion: Systematisches Umbenennen
$\lambda xM \xrightarrow{\alpha} \lambda y Sub_y^x[M], y \in FV(M)$
β-Reduktion (Ausrechnen)
$((\lambda xM)N) \xrightarrow{\beta} Sub_N^x[M]$, „call by name“(?)
$P \xrightarrow{*} P'$



Church-Rosser Term M ist in Normalform[2], wenn M nicht weiter reduzierbar ist (β-Reduktion).
Ein Term M hat eine Normalform, wenn $M \xrightarrow{*} N$, N ist in Normalform.

3.2 Eindeutigkeit von λ-Termen

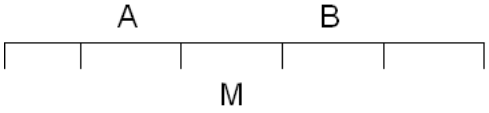
M = N : M und N haben die gleiche Normalform, oder M und N haben keine Normalform.

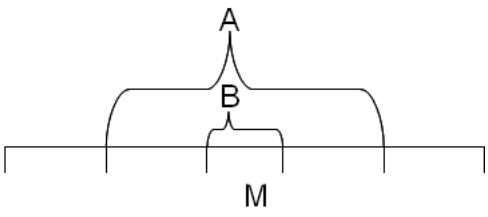
3.2.1 Fixpunkte:

(Realisierung von rekursiven Funktionen)
Für alle λ-Teile F gibt es einen λ-Term ϕ mit $(F\phi) = \phi$, ϕ heißt Fixpunkt.
Ein λ-Term M mit $MF = F(M, F)$ heißt Fixpunktkombinator $\Rightarrow$ MF ist stets Fixpunkt von F
Beispiel:
$Y \equiv AA$; $A \equiv \lambda xy.y(xxy)$
Eigenschaften von $Y : YF \equiv AAF \equiv F(AAF) \equiv F(yF)$
$Y \approx$ Curry's Paradoxer Kombinator
Frage: Auswahl einer guten („sicheren") Strategie beim Auswerten?
Gibt es eine sichere Strategie, die garantiert, dass ich zu einer Normalform komme, falls sie existiert?
Beispiel: $KI\Omega(\Omega \equiv ((\lambda x.xx)(\lambda x.xx)); I \equiv \lambda xx; K \equiv \lambda xy.x)$
Auswertung von Ω bricht nicht ab!

1. $KI\Omega \equiv KI((\lambda x.xx)(\lambda x.xx)) \xrightarrow{*} KI\Omega$
2. $KI\Omega \equiv \xrightarrow{\beta} I$

Redex ist ein Term, der β-Reduzierbar ist.
Ein Redex A ist links von einem Redex b in einem Term M, wenn A und B an verschiedenen Stellen stehen und es gilt
Eine Reduktionsfolge $M \xrightarrow{\beta*} N$ heißt Normalfolge, wenn bei jedem Reduktionsschritt das am





weitesten links stehende Redex reduziert wird.
Eine Reduktionsfolge $M \xrightarrow{\beta*} N$ heißt Standardreduktionsfolge, wenn das Reduzieren von links nach rechts erfolgt.

[2]Endergebnis

Beispiel:
$I \equiv \lambda xx$

1. $I(Ia) \equiv I((\lambda xx)a) \xrightarrow{\beta} Ia$
2. $I(Ia) \equiv (\lambda xx)(Ia) \xrightarrow{\beta} Ia$

zu:

1. ist keine Normalfolge
2. ist Normalfolge

beide sind Standardreduktionsfolgen reiner λ-Kalkül: $C = \emptyset$
Angewandter λ-Kalkül: $C \neq \emptyset$
Getypter λ-Kalkül
$\Rightarrow$ Theorie der abstrakten Datentypen

4 LISP

1962: Lisp-1.5: Handbuch

4.1 Daten

nur 1 Datentyp: S-Ausdrücke (s $\approx$ symbolisch)
S $\rightarrow$ atomare, S $\rightarrow$ nichtatomare
Atom: Name(Buchstaben,Ziffern)
A,Apfel, A123B
Nichtatomare S-Ausdrücke: Binärer Baum, Blätter $\approx$ Atome
Punktschreibweise(dot notation)
$< s - Ausdruck >::=< Atom > |(< s - Ausdruck > . < s - Ausdruck >)$

Beispiele: $(A.B), (A.(B.(C.D)))$
spezielle Baumstruktur
$(\alpha_1.(\alpha_2.(...(..\Omega))))$ (Entspricht einer Liste)
α_i: S-Ausdrücke
Ω: Endemarkierung
Liste $(\alpha_1...\alpha_n) \approx (\alpha_1.(\alpha_2.(...(\alpha_n.NIL)...)))$
2 Bedeutungen für NIL:

1. Endemarkierung
2. Representant der leeren Liste

Listen sind spezielle S-Ausdrücke
$< Liste >::= ()|NIL|(< Element > .. < Element >)$
$< Element >::=< S - Ausdruck > | < Liste >$

4.2 Basisfunktionen

1. **cons**
   Konstruiert aus: 2 S-Ausdrücken einen neuen S-Ausdruck
   $cons[\alpha;\beta] = (\alpha.\beta) \approx ..$
   $cons[A;B] = (A.B)$
   $cons[cons[A;B];C]$
   $= cons[(A.B);C] = ((A.B).C)$

2. **car** und **cdr**
   Selektion des S-Ausruckes α aus eine gegebenen S-Ausdruck β
   $$car[\beta] = \begin{cases} \beta_1 & \text{falls } \beta = (\beta_1.\beta_2)\text{(echter S-Ausdruck und kein Atom)} \\ \text{undef.} & \text{sonst} \end{cases}$$
   $$cdr[\beta] = \begin{cases} \beta_2 & \text{falls } \beta = (\beta_1.\beta_2)\text{(echter S-Ausdruck und kein Atom)} \\ \text{undef} & \text{sonst} \end{cases}$$
   Gegeben: Liste $(\beta_1..\beta_n) : car[(\beta_1...\beta_n)] = \beta_1$
   $cdr[(\beta_1...\beta_n)] = (\beta_2...\beta_n)$
   $car[((A1.A2).B)] = (A1.A2)$
   $cdr[((A1.A2).B)] = B$
   $car[(A\ B\ C)] = A; cdr[(A\ B\ C)] = (BC)$
   $car[(A)] = A$
   $cdr[(A)] = () = \text{NIL}$
   $car[A] = \text{undef}$
   Spezielle Abkürzung für Folgen von car's und cdr's
   $cadr[(A\ B\ C)] = car(cdr[A\ B\ C]) = B$
   $caddr$

4.3 Basisprädikate

1. **atom**
   Unterscheidet zwischen atomaren und nichtatomaren S-Ausdrücken
   $$atom[S] = \begin{cases} T & \text{S ist Atom} \\ F & \text{sonst} \end{cases}$$
   $atom[(U.V)] = F; \qquad atom[car[(U.V)]] = T$
   T,F spezielle Sonderatome

2. **eq**
   Überprüft Gleichheit von Atomen(!)
   $$eq[S_1;S_2] = \begin{cases} T & \text{falls } S_1, S_2 \text{ Atome und } S_1 = S_2 \\ F & \text{falls } S_1, S_2 \text{ Atome und } S_1 \neq S_2 \\ \text{undef} & \text{sonst} \end{cases}$$
   $eq[A;A] = T$
   $eq[A;B] = F$
   $eq[A;(AB)] = \text{undef}$

Bedingte Ausdrücke $[p_1 \to e_1; p_2 \to e_2; ...; p_n \to e_n]$
p_i Prädikate

e_i: Ausdrücke
Auswertung von links nach rechts, erstes Prädikate p_i mit $p_i = T$ liefert das Endergebnis $\underline{e_i}$

Abstraktion und Applikation Abstraktion $\lambda[[\underbrace{x_1, .., x_n}_{\text{Variablen}}]; \underbrace{E}_{\text{Rumpf}}]$
$\lambda[[u; v]; cons[u; v]]$
$\lambda[[x]; [atom[x] \to x; T \to cadr[x]]]$
Applikation von f auf $[a_1, .., a_n]$ erfolgt durch $f[a_1, .., a_n]$

Beispiel:

1. $\lambda[[u; v]; cons[u; v]]$[TASCHEN,RECHNER] $\to$ (TASCHEN.RECHNER)
   $\lambda[[u; v]; \lambda[[x; y]; cons[x; y]][car[v]; cdr[u]]][(A); (B)] \to (B)$
   Rekursive λ-Ausdrücke
   λ-Ausdrücke können durch das Sondersymbol, label' einen Namen erhalten (aus Kleinbuchstaben) $label[ff; \lambda[[x]; [atom[x] \to x; T \to ff[car[x]]]]]$

Funktionale $\lambda[[f]; f[f[(A\ B\ C)]]]$; braucht als Argument eine Funktion
z.B. $\lambda[[f]; f[f[(A\ B\ C)]]][cdr] \to (C)$
$\lambda[[x]; \lambda[[y]; cdr[cdr[cons[x; y]]]]]$ liefert als Ergebnis wieder eine Funktion: Funktion mit funktionalem Ergebnis.
$\lambda[[x]; \lambda[.....y]][A][(BC)]$ (2 Argumentgruppen)
$\to (C)$

4.4 Interpetierer (Lisp-Programm)

$\Rightarrow$ Eingaben müssen S-Ausdrücke sein
Eingabe $\begin{cases} \text{zu interpetierendes Programm} \\ \text{Startdaten (S-Ausdrücke)} \end{cases}$
1. Übersetzung von M-Programmen in S-Ausdrücke

1. $[arg_1; ...; arg_n] \to (f^*\ arg_1^*\ ...\ arg_n^*)$
2. S-Ausdrücke E $\to$ (QUOTE E)
3. Namen $\to$ Namen (groß)
4. bedingte Ausdruck $[p_1 \to e_1;\ ...\ ; p_n \to e_n] \to$ (COND $(p_1^*\ e_1^*)$ ... $(p_n^*\ e_n^*)$)
5. LAMBDA-Ausdruck $\lambda[[x_1;\ ...\ ; x_n]; E] \to$ (LAMBDA $(x_1$ ... $x_n)E^*$)
6. LABEL-Deklaration $label[f; e] \to$ (LABEL $f^*\ e^*$)
7. Funktionales Ergebnis bzw. funktionales Argument $f \to$ (FUNCTION f^*)

Beispiele: $car[x] \xrightarrow{1,3}$ (CAR X)
$T \xrightarrow{2.}$ (QUOTE T)
$label[ff; \lambda[[x]; atom[x] \to x; T \to car[x]]] \to$
(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (CAR X)))))

equal
null
append, cons
member
pairlis
assoc
sublis
evlis $= \lambda[[m;a];[null[m] \rightarrow$ NIL;
$T \rightarrow cons[eval[car[m];a];evlis[cdr[l];a]]]]$
$\lambda[[x;y];cons[car[x];y]][(A\ B);(C\ D)]$
1. evalquote [(LAMBDA (X Y) (CONS (CAR X) Y)); ((A B) (C D))]
$\rightarrow apply[...;.....;NIL]$
$\xrightarrow{8} eval[caddr[fn];pairlis[cadr[fn];x;a]]$
$= eval[caddr[$(LAMBDA (X Y) (CONS (CAR X) Y))$]$;
$\quad pairlis[cadr[$(LAMBDA (X Y) (CONS (CAR X) Y))$]$; ((A B) (C D)), NIL]]
$pairlis[...] \rightarrow ((X.(AB))(Y.(CD)))$
$eval[e;a] \equiv eval[$(CONS (CAR X) Y); ((X.(A B))(Y.(C D)))$]$
$\xrightarrow{19} apply[$car (CONS (CAR X) Y)$;evlis[CDR[$(CONS (CAR X) Y);a$];a]$
$\rightarrow cons[eval[$(CAR X); a$];evlis[$(Y);a$]]$
$\xrightarrow{\text{nur eval}} apply[car[$(CAR X)$];evlis[cdr[$(CAR X)$];a];a];$
$\rightarrow apply[CAR;[cons[eval[car[(X)];a;evlis[cdr[(X);a]];a]$
$\rightarrow apply[CAR;cons[cdr[sassoc[X;a;\lambda...];NIL;a]]]$
$\rightarrow A$
$\xrightarrow{\text{für evlis}} C\ D$
$\xrightarrow{\text{gesamt}} (A\ C\ D)$

5 Kombinatorische Logik

Schönfeld (1924), Curry (1930)
$Sfgx = fx(gx)$
$Kax = a$

$If = SKKf \rightarrow Kf(Kf) \rightarrow f$
Es gelten die Korollare von Church-Rosser

1. $y, \overline{y}$ Normalformen von $X \Rightarrow Y = \overline{Y}$
2. $X = Y \Rightarrow$ es existiert Z mit $X \xrightarrow{*} Z$ und $Y \xrightarrow{*} Z$
3. $X = Y$ und Y ist ein Normalfall $\Rightarrow X \xrightarrow{*} Y$
4. $X = Y \Rightarrow$ <u>entweder</u> X und Y haben dieselbe Normalform, <u>oder</u> X und Y haben beide <u>keine</u> Normalform.

$Sxyz = xz(yz)$, aber es gilt nicht $[x](Sxyz) = [x](xz(yz))$
$S(SS(Ky)(Kz)) \neq S(SI(Kz))(K(yz))$
Bei $(.)_\lambda$ bleiben Normalformen <u>nicht</u> erhalten.

$$\underbrace{SK}_{Normalform} \quad \Rightarrow (SK)_\lambda = (SK)$$

$$SK \equiv \underbrace{(\lambda xyz.xz(yz))K}_{\text{keine Normalform}} \rightarrow \lambda yz.Kz(yz) \rightarrow \lambda yz.z$$

6 FP-Systeme (Functional Programming)

Schema-Sprache, J.W.Backus (Vater von FORTRAN)

1. Identifiziere jedes Element einer Folge mit 1.
   $\overline{1}$ mit $\overline{1}$: $x = 1$
2. Identifiziere alle Elemente einer Folge mit 1.
   $\alpha\overline{1}$, $\alpha\overline{1} : \langle x_1, .., x_n\rangle = \langle 1, .., 1\rangle$
3. Addition zweier Einsen
   $+$
4. Addition aller Einsen in einer Einsenfolge
   $/+$
5. Verknüpfung der Teilprogramme + Name
   def länge $\equiv (/+) \circ (\alpha\overline{1})$
   länge $: \langle 1, 2, 3\rangle = (/+) \circ (\alpha\overline{1}) : \langle 1, 2, 3\rangle$
   $= (/+) : ((\alpha\overline{1}) : \langle 1, 2, 3\rangle)$
   $= (/+) : \langle 1, 1, 1\rangle)$
   $= + : \langle 1, /+ : \langle 1, 1\rangle\rangle$
   $= + : \langle 1, + : \langle 1, /+ : \langle 1\rangle\rangle\rangle$
   $= + : \langle 1, + \langle 1, 1\rangle\rangle$
   $= + : \langle 1, 2\rangle$
   $= 3$