Funktionale Programmierung

Prof. Dr. K. Indermark
Lehrstuhl für Informatik II
Rheinisch-Westfälische Technische Hochschule Aachen
Ahornstraße 55, 52072 Aachen
WWW: `http://www-i2.informatik.rwth-aachen.de/FP/`

WS 1995/96

`h.g.esser@gmx.de`
WWW: `http://home.pages.de/~hge/`

Inhaltsverzeichnis

Zu diesem Skript

Das vorliegende Skript wurde von mir auf der Grundlage einer Vorlesungsmitschrift aus dem Wintersemester 1995/96 erstellt. Dies ist kein offizielles Skript des Lehrstuhls. Zwar habe ich mich bei der Erstellung bemüht, Fehler meiner handschriftlichen Mitschrift zu korrigieren, kann aber natürlich keine Garantie für die Korrektheit übernehmen.

Dieses Skript darf nicht kommerziell, wohl aber privat und unentgeltlich weiterverbreitet werden (sofern nicht das Urheberrecht von Prof. Indermark dies verbietet), für Anregungen und Fehlerhinweise an meine Email-Adresse `h.g.esser@gmx.de` wäre ich dankbar.

Außer dem Skript zur Vorlesung Funktionale Programmierung habe ich auch Skripte zur Logikprogrammierung aus dem Sommersemester 1995 von Prof. Hanus und zur

Vorlesung Approximationstheorie I aus dem Wintersemester 1995/96 von Prof. Stens erstellt.

1 Einleitung

1.1 Imperative vs. deklarative Programmiersprachen

1.1.1 Imperative Programmiersprachen

Imperative Programmiersprachen: Fortran, Pascal, Modula 2, Ada, C, ...
Wertzuweisungen $x := y * x$ bezeichnen eine lokale Speichertransformation.
Ausführung eines Programmes: Folge lokaler Speichertransformationen
Prinzip des von-Neumann-Rechners, „von Neumann-Sprachen“
Software-Krise (70er Jahre) führte zur strukturierten Programmierung (Pascal), modularen Programmierung (Modula 2) und objektorientierten Programmierung (Oberon, N. Wirth).
J. Backus 1977: Entwicklung der *Funktionalen Programmierung* (FP). („von Neumannscher Flaschenhals der imperativen Programmiersprachen“)

1.1.2 Deklarative Programmiersprachen

Deklarative Programmiersprachen:
– logische Programmiersprachen (Prolog, Gödel)
– funktionale Programmiersprachen (LISP/Scheme, ML, Miranda, Haskell/Gofer)
– logisch-funktionale Programmiersprachen (Babel)
problemorientierte, rechnerunabhängige Definition von Funktionen und Relationen, also deterministischen und nicht-deterministischen Beziehungen von Objekten.

Merkmale funktionaler Programmiersprachen
– Funktionen als Elemente funktionaler Programme
– keine imperativen Variablen $\Longrightarrow$ keine Seiteneffekte
– Definition von Funktionen: Applikation, Abstraktion, Rekursion
– Funktionen höherer Ordnung, partielle Applikationen, Currying
– Listen; algebraische, polymorphe Datentypen
– Pattern Matching
– Berechnung durch Termreduktion; verzögerte Auswertung
– Referential Transparency: Werte von Teilausdrücken unabhängig von Berechnungsreihenfolge; Möglichkeit der parallelen Auswertung
– Automatische Speicherverwaltung

Vorteile funktionaler Programmiersprachen
– problembezogen, nicht maschinenorientiert
– kürzere, modulare Programme
– Vorteile bei Spezifikation von Software, Rapid Prototyping
– keine Seiteneffekte, Referential Transparency, parallele Auswertung

– klare mathematische Basis, Verifizierbarkeit, Sicherheit

1.2 Beispiele und Grundkonzepte funktionaler Programme (in Gofer)

1.2.1 Erste Beispiele

```
fact (n) = if n=0 then 1 else n*fact (n-1)
```
Bausteine: Konstanten: 0,1
Variablen: n
Grundfunktionen: $-, *, =$
Verzweigung: if ... then ... else ...

Komposition: Ausdrücke, Terme: $n * fact(n-1)$
rekursive Funktionsgleichung

Berechnungen erfolgen durch Termersetzung bzw. Reduktion (siehe Kapitel 3)

Andere Notation mit *Pattern Matching*:
```
 fact (0) = 1
fact (n+1) = (n+1) * fact(n)
```

In Gofer:
```
 factpm 0 = 1
factpm (n+1) = (n+1)*(factpm n)
```

Algebraischer Datentyp Nat:
```
data Nat = 0 | S Nat
```

Elemente von Nat: $0, S0, SS0, ..., S^n 0 \equiv n$
Erzeugung von Nat mit Konstruktoren 0, S.

Explizite Verzweigung mit Selektorfunktionen (z.B. (-1)) und Testfunktionen (z.B. (=0)). Pattern Matching eignet sich mehr für das Programmieren, Verzweigung ist besser für die Implementierung.

1.2.2 Listen als algebraischer Datentyp

```
data ListN = Nil | Cons Nat ListN
```

definiert Listen über natürlichen Zahlen. Konstruktoren sind Nil und Cons. Für Cons wird auch die Infix-Schreibweise `Nat:list` statt `Cons Nat list` benutzt. Außerdem schreiben wir auch `[]` statt `Nil` und z.B. `[2,3,4]` statt `2:3:4:[]`.

1.2.3 Pattern Matching auf Listen

```
lengthpm [] = 0
lengthpm (h:t) = 1 + (lengthpm t)
```

bzw. mit Verzweigung:
`lengthb l = if (null l) then 0 else 1 + (lengthb (tail l))`
mit Testfunktion null und Selektorfunktionen head, tail

1.2.4 Polymorphe Datentypen

`list (`α`) = [] | `α`:list(`α`)`
Gofer: `data List a = ..., [a] = List a`

Beispiel: `List Bool, List (List Int)`

Polymorphe Length-Funktion: `length :: List a` $\to$ `Int`

1.2.5 Binäre Bäume

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

Element: `Node 5 (Node 4 (Leaf 3) (Leaf 1)) (Node 2 (Leaf 1) (Leaf 6))`

Zahl der Blätter:

```
num-leafs (Leaf a) = 1
num-leafs (Node a t1 t2) = (num-leafs t1) + (num-leafs t2)
```

1.3 Funktionen höherer Ordnung

1.3.1 Currying

Vorbereitung: „Currying“ (Haskell P. Curry)
(Haskell: Referenzsprache der FP)

Elimination kartesischer Typen: $(A \times B) \to C$ durch $A \to (B \to C)$ ersetzen; dabei immer rechtsklammern.
Allgemein: $f : A^n \to B$ auffassen als $f : A \to A \to ... \to A \to B$.
Vorteil: partielle Applikation: $(f a_1 a_2)$ etc.

1.3.2 Funktionen, deren Argumente oder Werte selber Funktionen sind

```
sumi :: [Int] → Int
sumi [] = 0
sumi (h:t) = h + (sumi t)

prodi :: [Int] → Int
prodi [] = 1
prodi (h:t) = h * (prodi t)
```

Dies sind ähnliche Definitionen. Sie enthalten ein gemeinsames Definitionsmuster. Dies werden wir durch *funktionale Abstraktion* gewinnen, indem wir von (0,+) bzw. (1,*) abstrahieren:
Faltung `fold`:
`fold ::` $(a \to a \to a) \to a \to ([a] \to a)$
```
fold f c [] = c
fold f c (h:t) = h 'f' fold f c t
```

z.B. `sumi = fold (+) 0`, `prodi = fold (*) 1`, `anytrue = fold (or) False`, `alltrue = fold (and) True`

Funktionale Abstraktion ist ein mächtiges Konzept zur Modularisierung von Programmen.

Die Mehrfachauswertung von Teilausdrücken kann durch lokale Definitionen mittels *where*-Abstraktionen vermieden werden: Statt `f x = (... Ausdruck in x ...) * (... Ausdruck in x ...)` kürzer: `f x = y*y where y = (... Ausdruck in x ...)`

1.4 Der Quicksort-Algorithmus

Konkatenation von Listen:
```
append [] l = l
append (h:t) l = h : (append t l)
```
Infixnotation: `l++l' = append l l'` (in Gofer)
Herausfiltern einer Teilliste:

```
filter p [] = []
filter p (h:t) = if (p h) then h:filter p t else filter p t
```
Typ: `filter :: (a → Bool) → [a] → [a]`
Beispiel: `even n = if (n ‘mod‘ 2) == 0 then True else False`
`filter even [1,2,3,4,5]` gibt `[2,4]`.
Quicksort:
```
qs [] = []
qs (h:t) = qs (filter (<h) t) ++ [h] ++ qs (filter (>=h) t)
```

Partielle Applikation: ¡h :: Int → Bool

1.5 Unendliche Datenstrukturen

`list 1 = 1 : list 1`
definiert die unendliche Liste `[1,1,1,1,1, ... ]`.
`from n = n : from (n+1)`
definiert `[n,n+1,n+2, ...]`.

Anwendung: Primzahltest (Sieb des Erastosthenes):

```
filter’ p [] = []
filter’ p (h:t) = if (p h) then filter’ p t else h : filter’ p t
is-multiple :: Int → Int → Bool
is-multiple n m = (m ‘mod‘ n) == 0
sieve :: [Int] → [Int]
sieve (h:t) = h : sieve (filter’(is-multiple h) t)
member :: Int → [Int] → Bool
member n [] = False
member n (h:t) = (n==h) || (if h>n then False else member n t)
is-prime :: Int → Bool
is-prime n = member n (sieve (from 2) )
```

erledigt den Primzahltest.
(lazy evaluation: member durchsucht die Liste nur so weit, wie nötig. Call by need)

2 Terme und ihre Berechnung

Terme (Ausdrücke, „expressions“) bilden ein Grundkonzept der funktionalen Programmierung. Sie werden aus Konstanten und Variablen durch Applikation von Grundfunktionen oder Benutzerfunktionen (d. h. benutzerdefinierten Funktionen) erzeugt.
Hier: Terme erster Ordnung, noch keine Funktionen höherer Ordnung.

Anwendungen:

- Definition von Benutzerfunktionen:
  $F(x) = (3x - 7) * (4x - 1)$

- Programmaufruf / Eingabe
  $(3 + F(12)) * F(3)$

- Algebraische Datentypen als Konstruktorterme
  CONS (3, CONS (2, CONS (1,NIL)))
  1 : 2 : 3 : []
  $[1, 2, 3]$

Fragen:

- Syntax: Was haben folgende Ausdrücke gemeinsam?
  (x-2)*(y-2)
  - x 2 - y 2
  x 2 - y 2 - *
  Die algebraische Syntax ist darstellungsunabhängig; abstrakte Syntax vs. konkrete Syntax. (wichtig im Compilerbau)
- Semantik: sequentielle und parallele Berechnungen. Algebraische Semantik, berechnungsunabhängig
- Implementierung: Kellertechnik, Stackcode

2.1 Algebraische Grundlagen

Ziel: algebraische Syntax und Semantik von Termen, algebraische Datentypen (später).

Grundlage für Berechnungen:

- Sorten als Bezeichnungen von Basismengen
- Operationssymbole als Bezeichnungen von Basisoperationen

Algebraisch: Signatur (Syntax), Algebra (Semantik)

Sortierte Mengen
Sei S eine nicht-leere Menge von Sorten. Wenn A eine Menge ist und $\sigma : A \to S$, dann heißt A S-sortiert (durch σ). Für $s \in S$ bezeichnet $A^s := \sigma^{-1}(s)$ alle Elemente von A der Sorte s. Elemente $a \in A^s$ (mit $\sigma(a) = s$) bezeichnet man auch durch a^s. Seien A und B S-sortierte Mengen und $f : A \to B$. Dann heißt f *sortentreu*, falls $f(A^s) \subseteq f(B^s)$.

Generalvoraussetzung: Im folgenden werden Abbildungen zwischen sortierten Mengen stets als sortentreu vorausgesetzt.

Definition (Signatur)
Sei S eine nicht-leere Menge von Sorten und Ω eine $S^* \times S$-sortierte Menge von *Operationssymbolen*. Dann heißt $\Sigma = \langle S, \Omega \rangle$ eine *Signatur*. Wenn $f \in \Omega^{(w,s)}$, dann heißt

- $s_1 ... s_n$ die Folge der *Argumentsorten*,

- s die *Zielsorte*,
- (w, s) der Typ von f.

Spezialfall: Wenn $n = 0$, also $w = \varepsilon$, so heißt $f^{(\varepsilon,s)}$ *Konstantensymbol* vom Typ s. Signaturen mit $|S| = 1$ heißen *einsortig* oder *homogen*, sonst *mehrsortig* oder *heterogen*. Man schreibt dann (n) statt (s^n, s) und nennt n die *Stelligkeit*.

Beispiel: Signatur
(** Folie 2.1 **)

In algebraische Spezifikationssprache: Aufschlüsseln nach Zielsorten, also z.B.
ops bool = (true, false, empty(stack))
(In Klammern stehen die Argumentsorten.)
Betrachte diese Notation als CFG. (kontextfreie Grammatik) ($\rightsquigarrow$ rewriting systems, software engineering).

Graphische Darstellung der stack-Signatur
(** Folie 2.2 **)

Definition (Operation)
Sei $\Sigma = \langle S, \Omega \rangle$ eine Signatur, $A = \bigcup_{s \in S} A^s$ eine S-sortierte Menge. Dann bezeichne

$$A^\varepsilon := \{\emptyset\}, A^{s_1 \ldots s_n} := A^{s_1} \times \ldots \times A^{s_n}, s_1, \ldots, s_n \in S$$

$Ops^{(w,s)}(A) := \{f \mid f : A^w \to A^s\}, Ops(A) := \cup\{Ops^{(w,s)} \mid (w, s) \in S^* \times S\}$.
$f : A^w \to A^s$ heißt *Operation auf A vom Typ(w, s)*. . Falls $w = \varepsilon$, so heißt f eine *Konstante* vom Typ s.
Da $A^\varepsilon = \{\emptyset\}$, wird $Ops^{(\varepsilon,s)}(A)$ auch mit A^s identifiziert: statt $f()$ schreibt man auch f.

Definition (Σ-Algebra)
Sei $\Sigma = \langle S, \Omega \rangle$ eine Signatur, A eine S-sortierte Menge und $\alpha : \Omega \to Ops(A)$ (α sortentreu). Dann heißt $\mathcal{A} = \langle A; \alpha \rangle$ eine *Σ-Algebra*. Wir schreiben auch $f_\mathcal{A}$ statt $\alpha(f)$.

Beispiel: Algebra
(** Folie 2.3 **)

Definition (Homomorphismus)
Seien $\Sigma = \langle S, \Omega \rangle$ eine Signatur und $\mathcal{A} = \langle A; \alpha \rangle, \mathcal{A}' = \langle A', \alpha' \rangle$ Σ-Algebren. Dann heißt eine Abbildung $h : A \to A'$ ein *Homomorphismus*, falls für alle $(w, s) \in S^* \times S, f \in \Omega^{(w,s)}$ und $a = (a_1, \ldots, a_n) \in A^w; w = (s_1, \ldots, s_n)$ gilt:

$$h(f_\mathcal{A}(a_1, \ldots, a_n)) = f_{\mathcal{A}'}(h(a_1), \ldots, h(a_n)).$$

Definition (Σ-Termalgebra)
Sei $\Sigma = \langle S, \Omega \rangle$ eine Signatur und $X = \bigcup_{s \in S} X^s$ eine S-sortierte Menge von *Variablen*.

Die Σ-*Termalgebra* (syntaktische Algebra) über X, $\mathcal{T}_\Sigma(X) = \langle T_\Sigma(X); \alpha_T\rangle$ ist definiert durch (i) und (ii):
(i) $T_\Sigma(X) = \bigcup\{T_\Sigma(X)^s \mid s \in S\}$ ist induktiv bestimmt durch

- $X^s \cup \Omega^{(\varepsilon,s)} \subseteq T_\Sigma(X)^s$ (Variablen und Konstanten vom Typ s)
- Wenn $f \in \Omega^{(s_1...s_n,s)}$ und $t_i \in T_\Sigma(X)^{s_i}$, so ist $ft_1...t_n \in T_\Sigma(X)^s$. (Applikation)

(ii) α_T ordnet jedem Operationssymbol $f \in \Omega^{(s_1...s_n,s)}$ eine *Termoperation* zu:

$$\alpha_T(f) : T_\Sigma(X)^{s_1} \times ... \times T_\Sigma(X)^{s_n} \to T_\Sigma(X)^s : (t_1, ..., t_n) \mapsto ft_1...t_n$$

Satz
Die Σ-Termalgebra $\mathcal{T}_\Sigma(X)$ ist *frei von X erzeugt*, d.h.
(i) Jedes Element aus $T_\Sigma(X)$ ist mit endlich vielen Termoperationen erzeugbar,
(ii) Freiheit: Wenn $\mathcal{A} = \langle A; \alpha\rangle$ eine Σ-Algebra und $\beta : X \to A$ eine Belegung der Variablen ist, so läßt sich β auf genau eine Weise zu einem Homomorphismus $\widehat{\beta} : \mathcal{T}_\Sigma(X) \to \mathcal{A}$ fortsetzen. ($\leadsto$ Reduktionssatz von Dedekind über induktiv definierte Abbildungen, 1888)

Beweisidee: eindeutige Zerlegbarkeit von Termen in Teilterme ($\leadsto$ ermöglicht Pattern Matching: Aufbau der Terme durch Konstruktoren, Zerlegung durch Selektoren)

Satz
Frei von X erzeugte Σ-Algebren sind isomorph.
(Compilerbau: Syntaxanalyse = Strukturerkennung)

Beweisidee: Komposition von Homomorphismen ergibt Identität. Es folgt: Homomorphismen sind injektiv und surjektiv, also bijektiv.
Beweis: Seien $\mathcal{T}_\Sigma(X), \mathcal{T'}_\Sigma(X)$ zwei von X frei erzeugte Σ-Algebren. Die Identität $i : X \to X$ hat eindeutige Fortsetzungen $f : \mathcal{T}_\Sigma(X) \to \mathcal{T'}_\Sigma(X)$ bzw. $g : \mathcal{T'}_\Sigma(X) \to \mathcal{T}_\Sigma(X)$. Nun sind $g \circ f, id_{\mathcal{T}_\Sigma(X)} : \mathcal{T}_\Sigma(X) \to \mathcal{T}_\Sigma(X)$ beides Fortsetzung von i. Wegen der Eindeutigkeit folgt $g \circ f = id, f \circ g = id$. Damit sind f und g bijektiv, also Isomorphismen, d.h. die beiden frei erzeugten Algebren sind isomorph.

Abstrakte Syntax: Isomorphie frei erzeugter Algebren erlaubt *darstellungsunabhängige* Auffassung syntaktischer Objekte. Wesentlich ist die Struktur der Objekte, nicht ihre Darstellung. Somit ist ein Wechsel der Darstellung möglich ($\leadsto$ Compilerbau).

Folgerung
Für die Semantik eines Terms (oder die Code-Generierung) ist nur sein struktureller Aufbau, nicht seine Darstellung relevant.

Konkrete Syntax
$T_\Sigma(X)$: klammerfreie Präfixnotation. Alternativ: Postfixnotation, Klammern und

Kommata, Infixnotation bei binären Operationen, Mixfixnotation (z.B. if ... then ... else).

Algebraische Semantik
Eindeutige Fortsetzung von Variablenbelegungen; berechnungsunabhängige Semantik. $t \in T_\Sigma(X), \beta : X \to A, \mathcal{A}$,

- $\widehat{\beta}(x) = \beta(x)$ für $x \in X$
- $\widehat{\beta}(ft_1...t_n) = f_{\mathcal{A}}(\widehat{\beta}(t_1), ...\widehat{\beta}(t_n))$

(Das genau ist Homomorphie.) $\widehat{\beta}$ ist induktiv definiert.

Definition (algebraische, initiale Semantik)
(i) Für $t \in T_\Sigma(X)$, eine Algebra $\mathcal{A} = \langle A; \alpha \rangle$ und eine Belegung $\beta : X \to A$ heißt

$$\mathcal{M}^{alg}_{(\mathcal{A},\beta)}[t] := \widehat{\beta}(t) \in A$$

die *algebraische Semantik* von t bzgl. $\mathcal{A}$ und β.
(ii) Spezialfall: $X = \emptyset$, Grundterme $\mathcal{T}_\Sigma$ statt $\mathcal{T}_\Sigma(\emptyset)$, $h_{\mathcal{A}} : \mathcal{T}_\Sigma \to \mathcal{A}$. Dann heißt $\mathcal{M}^{alg}_{\mathcal{A}}[t] := h_{\mathcal{A}}(t)$ die *initiale Semantik* von t bzgl. $\mathcal{A}$.

2.2 Reduktionssemantik für Terme

Gegeben: $\Sigma = \langle S, \Omega \rangle, X = (X^s \mid s \in S), \mathcal{A} = \langle A; \alpha \rangle, \beta : X \to A$. Für $t \in T_\Sigma(X)$ ist $\mathcal{M}^{alg}_{(\mathcal{A},\beta)}[t]$ induktiv definiert – unabhängig von Berechnungen.

Reduktionssemantik: konfluentes Termersetzungssystem, nicht-deterministisch; deterministische Auswertungsstrategien: sequentiell / parallel.

Definition (Reduktionsrelation)

1. *Berechnungsterme* $Comp := T_\Sigma(A)$ (A statt X !)
2. *Reduktionsregeln*:
   $fa_1...a_n \to f_{\mathcal{A}}(a_1, ..., a_n) \in A^s$ für $f \in \Omega^{(s_1...s_n,s)}, a_i \in A^{s_i}$
3. *Reduktionsrelation*: $\Rightarrow \subseteq Comp^2$ ist durch Induktion über die Termstruktur definiert:
   (a) Wenn $t \to t'$ Reduktionsregel, dann $t \Rightarrow t'$
   (b) Für jedes $t \in Comp$ gilt: $t \Rightarrow t$
   (c) Wenn $ft_1...t_n \in Comp$ und $t_i \Rightarrow u_i$ für $1 \leq i \leq n$, dann $ft_1...t_n \Rightarrow fu_1...u_n$

Beispiel: Reduktionen für arithmetische Ausdrücke
(** Folie 2.4 **)

Bemerkung: Das Verhalten ist nicht-deterministisch, sowohl sequentielle als auch parallele Reduktion sind möglich.

Reduktionssemantik
$t \in T_\Sigma(X)$ bestimmt bezüglich $\beta : X \to A$ einen Berechnungsterm $t_\beta \in T_\Sigma(A)$ durch Einsetzen:
$\widetilde{\beta} : X \to T_\Sigma(A)$
$\widehat{\widetilde{\beta}} : T_\Sigma(X) \to T_\Sigma(A), t_\beta := \widehat{\widetilde{\beta}}(t)$

Definition und Satz (Reduktionssemantik)

1. Für jeden Berechnungsterm $t \in Comp$ gibt es genau ein $a \in A$ mit $t \Rightarrow^* a$.
   Für $t \in T_\Sigma(X)$ definieren wir die *Reduktionssemantik* von t bezüglich $\mathcal{A}$ und β durch
   $$\mathcal{M}^{red}_{(\mathcal{A},\beta)}[t] = a \text{ :gdw } t_\beta \Rightarrow^* a$$
2. $\mathcal{M}^{red}_{(\mathcal{A},\beta)}[t] = \mathcal{M}^{alg}_{(\mathcal{A},\beta)}[t]$

Beispiel: Linksreduktion
(** Folie 2.5 **)

Beispiel: Parallele Reduktion
(** Folie 2.6 **)

2.3 Stackimplementierung für Terme

Stackprinzip (Keller, Stapel) von Bauer, Samelson (ca. 1958)
Die Auswertung arithmetischer Ausdrücke durch Übersetzung von Termen in Stackcode, sequentielle Implementierung

Stackmaschine
(** Zeichnung von Seite 21 **)

Zustandsraum: $ZR := DK \times HS$
mit **Datenkeller** $DK := A^*$ (Spitze rechts)
und **Hauptspeicher** $HS := \{\ \beta \mid \beta : X \to A\ \}$
(Identifiziere Speicher mit Belegungsfunktion)

Befehlssatz
$Cmd := \{\ LOAD\ i \mid 1 \leq i \leq n\ \} \cup \{\ EXEC\ f \mid f \in \Omega\ \}$

Programme
$Prog := \{ \Gamma_1; ...; \Gamma_p \mid \Gamma_i \in Cmd, p \geq 1 \}$
(Straight line code: no jumps, no branch)

Befehlssemantik
$\Gamma \in Cmd \mapsto \mathcal{C}[\Gamma] : ZR - \to ZR$ (partiell)
mit
$\mathcal{C}[LOAD\ i](d, \beta) := (d \cdot \beta(x_i), \beta)$,
$\mathcal{C}[EXEC\ f](d \cdot a_1 \cdot ... \cdot a_n, \beta) := (d \cdot a, \beta)$, falls $f_{\mathcal{A}}(a_1, ..., a_n) = a$.

Programmsemantik
Für $P = \Gamma_1; ...; \Gamma_p \in Prog$ ist $\mathcal{M}[P] : ZR - \to ZR$ definiert durch

$$\mathcal{M}[\Gamma_1; ...; \Gamma_p] := \mathcal{C}[\Gamma_p] \circ ... \circ \mathcal{C}[\Gamma_1].$$

Beispielrechnung der Stackmaschine
(** Folie 2.7 **)

Definition (Übersetzung von Termen in Stackcode)
$trans : T_\Sigma(X) \to Prog$ sei induktiv definiert durch
$trans(x_i) := LOAD\ i$
$trans(ft_1...t_n) := trans(t_1); trans(t_2); ...; trans(t_n); EXEC\ f$

Beziehung zwischen Semantik und Codegenerierung
$\widehat{\beta}(x_i) = \beta(x_i), \widehat{\beta}(ft_1...t_n) = f_{\mathcal{A}}(\widehat{\beta}(t_1), ..., \widehat{\beta}(t_n))$.

Satz (Korrektheit der Übersetzung)
Für $t \in T_\Sigma(X)$ gilt bezüglich $\mathcal{A}$ und β:

$$\mathcal{M}^{alg}_{\mathcal{A},\beta}[t] = a \text{ gdw } \mathcal{M}[trans(t)](\varepsilon, \beta) = (a, \beta)$$

Beweis: McCarthy 1964, erster Compiler-Korrektheitsbeweis

3 Rekursive Funktionsdefinition erster Ordnung

Verwendung von Termen:

- imperativ: $x := 3x + 7$
- funktional: $F(x) = 3x + 7$

3.1 Explizite Funktionsdefinition

Syntax: $\Sigma = \langle S, \Omega \rangle, X = \{x_1, ..., x_n\}$ S-sortiert, $t \in T_\Sigma(X)$

Polynom: $F(x_1, ..., x_n) = t$ „funktionale Abstraktion“ , $F = \lambda(x_1, ..., x_n).t$ (λ-Abstraktion; später)

Semantik
Eine Σ-Algebra $\mathcal{A} = \langle A; \alpha \rangle$ bestimmt für obiges Polynom F mit den Sorten $\sigma(x_i) = s_i$ und $\sigma(t) = s$ die *Polynomfunktion* $\mathcal{M}_\mathcal{A}[F] : A^{s_1} \times ... \times A^{s_n} \to A^s$ mit

$$\mathcal{M}_\mathcal{A}[F](a_1, ..., a_n) = \mathcal{M}_{(\mathcal{A},\beta)}[t]$$

mit $\beta(x_i) = a_i, 1 \leq i \leq n$).

Spezielle Polynomfunktionen
(solche von atomaren Termen)

- die *Projektionen* $proj_i^{(n)} := \mathcal{M}_\mathcal{A}[\lambda(x_1, ..., x_n).x_i]$
- die *konstanten Funktionen* $const^{(n)}(c_\mathcal{A}) := \mathcal{M}_\mathcal{A}[\lambda(x_1, ..., x_n).c]$

Programmiertechnik
Entschachtelung von Termen durch Entschachtelung von Funktionsdefinitionen.
Beispiel: $F(x, y) = 3(x^2 - y^2)(x^3 + y^3)$ kann ersetzt werden durch
$F(x, y) = 3G(x, y)H(x, y)$
$G(x, y) = x^2 - y^2$
$H(x, y) = x^3 + y^3$.

3.2 Rekursion und Fixpunktsemantik

3.2.1 Monotone Funktionen auf Halbordnungen

Beispiel 1:
$F(x) = G(H(x))$
$G(x) = 1$
$H(x) = H(x)$
(G ist Konstante, H die nirgends definierte Funktion.) Frage: $F(0) = 1$ oder $F(0)$ undefiniert?

Operationelle Semantik:
Zwei Möglichkeiten:

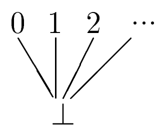
1. „call by value“ : $F(0) \Rightarrow G(H(0)) \Rightarrow G(H(0)) \Rightarrow ...$
2. „call by name“ : $F(0) \Rightarrow G(H(0)) \Rightarrow 1$

Denotationelle Semantik:
$f = g \circ h$ mit $g = \lambda x.1$, h nirgends definiert. Problem: *Komposition partieller Funktionen.* Call-by-name-Auswertung zeigt: $G(H(0))$ ist definiert, obwohl $H(0)$ nicht definiert ist.

Lösung: Einführung eines neuen Elementes $\perp$ („bottom“) . $\widehat{\mathbb{N}} := \mathbb{N} \cup \{\perp\}$ (auch $N_\perp$).
Partielle Funktion $f : \mathbb{N} - \rightarrow \mathbb{N}$ erweitern zu totaler Funktion $f' : \mathbb{N} \rightarrow \widehat{\mathbb{N}}$ mit $f'(a) = \perp$, falls $f(a)$ nicht definiert, und wegen Komposition zu $\widehat{f} : \widehat{\mathbb{N}} \rightarrow \widehat{\mathbb{N}}$.
Entscheidend ist nun, welchen Wert $\widehat{f}(\perp)$ annehmen soll.
Für die konstante Funktion $g = \lambda x.1$ gibt es zwei Möglichkeiten:
a) *strikte* Erweiterung: $\widehat{g}(\perp) = \perp$ entspricht call-by-value.
b) *nichtstrikte* Erweiterung: $\widehat{g}(\perp) = 1$ entspricht call-by-name.
Fall a) entspricht der üblichen mathematischen Komposition partieller Funktionen, Fall b) wird möglich, wenn der Funktionswert vom Argument unabhängig ist (für einstellige Funktionen: konstant).

Mathematische Charakterisierung strikter und konstanter Funktionen von $\widehat{\mathbb{N}}$ nach $\widehat{\mathbb{N}}$: Erkläre auf $\widehat{\mathbb{N}}$ eine Halbordnung durch

$$a \leq b \text{ :gdw } a = b \vee a = \perp$$



Dann gilt für $f : \widehat{\mathbb{N}} \rightarrow \widehat{\mathbb{N}}$:
f strikt oder konstant gdw. f *monoton*, d.h. $a \leq b \Rightarrow f(a) \leq f(b)$.
(Scott 196?: $\leq$ entspricht „weniger Informationsgehalt“)
Beweis: „$\Rightarrow$“ : klar.
„$\Leftarrow$“ : f monoton. Sei f nicht strikt (zeige also: f konstant). $f(\perp) = n \in \mathbb{N}$ (sonst wäre f strikt). $\perp \leq a \Rightarrow f(\perp) \leq f(a) \Rightarrow f(a) = n$, d.h. f ist konstant.

Ergebnis: $[\widehat{\mathbb{N}} \rightarrow \widehat{\mathbb{N}}] := \{ f \mid f : \widehat{\mathbb{N}} \rightarrow \widehat{\mathbb{N}} \text{ monoton} \}$ ist ein geeigneter semantischer Bereich. (Bemerk.: später „stetig“ statt „monoton“ ; hier fallen die Begriffe stetig und monoton zusammen.)

Vorteil: statt partieller Funktionen mit verschiedenen Kompositionen jetzt totale, monotone Funktionen.

3.2.2 Stetige Funktionen auf vollständigen Halbordnungen

Beispiel 2: Fakultät
F(x) = if x=0 then 1 else x*F(x-1)   bzw.
F = λ x.if x=0 then 1 else x*F(x-1)

Diese Funktion bestimmt das *Gleichungs-Funktional* bzw. *Einsetzungsfunktional* $\Phi : [\widehat{\mathbb{N}} \rightarrow \widehat{\mathbb{N}}] \rightarrow [\widehat{\mathbb{N}} \rightarrow \widehat{\mathbb{N}}]$ durch $\Phi(g) = \lambda x.$ if $x = 0$ then 1 else $x * g(x-1)$.
Dann ist $fac : \widehat{\mathbb{N}} \rightarrow \widehat{\mathbb{N}}$ mit $fac(n) = n!$ ein Fixpunkt von Φ: $\Phi(fac) = fac$; also ist fac eine Lösung der rekursiven Funktionsgleichung.

Mathematische Charakterisierung der Lösung
Halbordnung auf $[\widehat{\mathbb{N}} \to \widehat{\mathbb{N}}]$:

$$f \leq g \quad \text{:gdw } f(a) \leq g(a) \quad \forall a \in \widehat{\mathbb{N}}$$

(entspricht Inklusion der Funktionsgraphen; wenn die Graphen keine Kanten $a \to \bot$ enthalten.)

1. $\langle [\widehat{\mathbb{N}} \to \widehat{\mathbb{N}}]; \leq \rangle$ ist eine vollständige Halbordnung. $f_0 \leq f_1 \leq f_2 \leq \ldots$ besitzt eine kleinste obere Schranke $\bigsqcup_{i \in \mathbb{N}} f_i$.

2. Φ ist stetig: $\Phi(\bigsqcup_{i \in \mathbb{N}} f_i) = \bigsqcup_{i \in \mathbb{N}} \Phi(f_i)$

3. Fixpunkteigenschaft: Φ besitzt einen kleinsten Fixpunkt: $\bigsqcup_{i \in \mathbb{N}} \Phi^i(\lambda x.\bot)$.

Für das Beispiel ergibt sich:
$\Phi^1(\lambda x.\bot) = \lambda x.\ \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x * (\lambda x.\bot)(x-1) = \lambda x.\ \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ \bot$
$\Phi^2(\lambda x.\bot) = \lambda x.\ \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x * (\lambda x.\ \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ \bot)(x-1) = \lambda x.\ \underline{\text{if}}\ x \in \{0,1\}\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ \bot$
Allgemein:
$\Phi^k(\lambda x.\bot) = \lambda x.\ \underline{\text{if}}\ x < k\ \underline{\text{then}}\ k!\ \underline{\text{else}}\ \bot$

Ergebnis: Stetige Funktionen auf vollständigen Halbordnungen.

Beachte: $\widehat{\mathbb{N}}$ ist flache Halbordnung (maximale Kettenlänge 2). Daraus folgt: $\widehat{\mathbb{N}}$ ist vollständig, und für $f : \widehat{\mathbb{N}} \to \widehat{\mathbb{N}}$ gilt: f ist genau dann monoton, wenn f stetig ist.

3.3 Vollständige Halbordnungen, stetige Funktionen, Fixpunktsatz

Definition (Halbordnung)
Sei $A \neq \emptyset$ und $\leq \subseteq A^2$, so daß $\leq$

- reflexiv: $a \leq a$
- transitiv: $a \leq b$ und $b \leq c \Rightarrow a \leq c$
- antisymmetrisch: $a \leq b$ und $b \leq a \Rightarrow a = b$

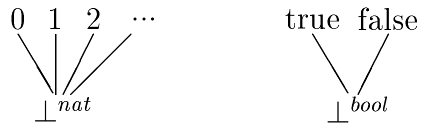
für alle $a, b, c \in A$. Dann heißt $\mathcal{A} = \langle A; \leq \rangle$ eine *Halbordnung*.

Beispiel:

- $\langle \widehat{\mathbb{N}}; \leq \rangle$ flache Halbordnung
- $\langle [\widehat{\mathbb{N}}, \widehat{\mathbb{N}}]; \leq \rangle$

- $\langle Pot(\Sigma^*); \subseteq \rangle$, formale Sprachen
  (Automaten: $\delta(q_i, a) = q_j \Rightarrow L_i = aL_j$)

Ist A S-sortiert, so erweitern wir jedes A^s um ein eigenes Bottom-Element $\perp^s$: $\widehat{A} := \bigcup_{s \in S} \widehat{A^s}$ mit $\widehat{A^s} := A^s \cup \{\perp^s\}$.
Beispiel:



Definition (monotone Funktion)
Seien $\mathcal{A}_i = \langle A_i; \leq_I \rangle$ für $i = 1, 2$ Halbordnungen und $f : A_1 \to A_2$. f heißt *monoton*, falls $a \leq_1 b \Rightarrow f(a) \leq_2 f(b) \quad \forall a, b \in A_1$.

Folgerung: Monotone Funktionen sind unter Komposition abgeschlossen.

Definition (gerichtete Menge)
Sei $\langle A; \leq \rangle$ eine Halbordnung, $T \subseteq A$ mit $T \neq \emptyset$. Dann heißt T *gerichtet*, falls $\forall\ a, b \in T\ \exists\ c \in T : a, b \leq c$.

Beispiel: Ketten wie etwa $\{\ \Phi^i(\lambda x.\perp) \mid i \in I\!N\ \}$ (Beispiel 2) sind gerichtete Mengen.

Folgerung: Gerichtete Mengen sind unter monotonen Funktionen abgeschlossen: Wenn T gerichtet und f monoton ist, dann ist $f(T)$ gerichtet.

Definition (kleinste obere Schranke)
Sei $\langle A; \leq \rangle$ eine Halbordnung, $T \subseteq A, a \in A$. Dann heißt a *obere Schranke* von T, falls $T \leq a$. a heißt *kleinstes Element* von T, falls $a \leq T$ und $a \in T$. Besitzt $\{\ b \mid T \leq b\ \}$ ein kleinstes Element, so heißt dies *kleinste obere Schranke (Supremum)* $\bigsqcup T$ von T.

Definition (vollständige Halbordnung)
Eine Halbordnung $\mathcal{A} = \langle A; \leq \rangle$ heißt *vollständig*, falls gilt:

1. $\mathcal{A}$ besitzt ein kleinstes Element $\perp_{\mathcal{A}} \in A$,
2. Jede gerichtete Menge $T \subseteq A$ hat in A eine kleinste obere Schranke $\bigsqcup T \in A$.

Bemerkung: Hätte man in der Definition von 'gerichtet' auch $T = \emptyset$ zugelassen, dann wäre 1) überflüssig gewesen mit $\perp_{\mathcal{A}} = \bigsqcup \emptyset$.)

Beispiele:

- $\langle Pot A; \subseteq \rangle$ ist vollständige Halbordnung mit $\perp = \emptyset$ und $\bigsqcup T = \bigcup T$ (sogar vollständiger Verband).
- Die flache Halbordnung $\langle I\!N; \leq \rangle$ ist auch vollständig: kleinstes Element ist $\perp \in \widehat{I\!N}$, und die einzigen geordneten Mengen sind $\{\perp\}, \{n\}, \{\perp, n\}$ (für $n \in I\!N$).

- $\langle[\widehat{I\!N} \to \widehat{I\!N}]; \leq\rangle$ ist ebenfalls vollständig: kleinstes Element ist $\lambda x.\bot$, und $\bigsqcup\{ f_i \mid i \in I \} := \lambda x. \bigsqcup_{i\in I} f_i(x)$.

Definition (stetige Abbildung)
Seien $\mathcal{A}_i = \langle A_i; \leq_I\rangle$ für $i = 1,2$ vollständige Halbordnungen. Eine Abbildung $f : A_1 \to A_2$ heißt *stetig*, falls f monoton ist und für jede gerichtete Teilmenge $T \subseteq A_1$ gilt:

$$f(\bigsqcup T) = \bigsqcup f(T)$$

Bemerkung: Für gerichtete Mengen T ist auch $f(T)$ gerichtet, weil f monoton ist.

Bemerkung: Hier sieht man eine Auswirkung der Tatsache, daß die leere Menge *nicht* gerichtet ist: Denn wäre sie es doch, dann würde für jede stetige Abbildung f gelten, daß $f(\bot) = \bot$!

Folgerung: Stetige Abbildungen sind unter Komposition abgeschlossen.

Satz: Fixpunktsatz von Tarski
Sei $\mathcal{A} = \langle A; \leq\rangle$ eine vollständige Halbordnung und $f : A \to A$ stetig. Dann besitzt f einen kleinsten Fixpunkt in A, nämlich

$$fix(f) := \bigsqcup\{ f^i(\bot) \mid i \in I\!N \}$$

Beweis:

1. Existenz: Es gilt $\bot \leq f(\bot) \leq f^2(\bot) \leq ....$ Daher ist $\{ f^i(\bot) \mid i \in I\!N \}$ eine Kette, welche eine obere Schranke besitzt, da $\mathcal{A}$ vollständig ist.

2. $f(\bigsqcup_{i\in I\!N} f^i(\bot) \underbrace{=}_{f \text{ stetig}} = \bigsqcup_{i\in I\!N} f(f^i(\bot)) = \bigsqcup_{i\in I\!N} f^{i+1}(\bot) = \bigsqcup_{i\in I\!N} f^i(\bot)$, also ist $fix(f)$ Fixpunkt.

3. $fix(f)$ ist kleinster Fixpunkt von f: Sei a ein Fixpunkt von f. Dann gilt: $a = f(a) = f^i(a)$ und $\bot \leq a$. Aus der Monotonie von f folgt: $f^i(\bot) \leq f^i(a) = a$, also $f^i(\bot) \leq a$ für alle $i \in I\!N$, woraus aber $fix(f) = \bigsqcup_{i\in I\!N} f^i(a) \leq a$ folgt (a obere Schranke).

Anwendung: Lösung eines rekursiven Gleichungssystems als Fixpunkt des entsprechenden Gleichnungsfunktionals

Satz und Definiton (Funktionenraum, Produktraum)
Seien $\mathcal{A}_i = \langle A_i; \leq_I\rangle$ für $i = 1,2$ vollständige Halbordnungen. Dann gilt:

1. Der *Funktionenraum* $[\mathcal{A}_1 \to \mathcal{A}_2] := \langle\{ f : A_1 \to A_2 \text{ stetig } \}; \leq\rangle$ mit

$$f \leq g \quad :\text{gdw.} \quad f(a) \leq_2 g(a)\ \forall a \in A_1$$

   sowie

2. Der *Produktraum* $\mathcal{A}_1 \times \mathcal{A}_2 := \langle A_1 \times A_2; \leq \rangle$ mit

$$(a_1, a_2) \leq (b_1, b_2) \ \text{ : gdw. } \quad a_i \leq_i b_i \ (i = 1, 2)$$

sind vollständige Halbordungen.

Anwendung: Sei A eine S-sortierte Menge. Dann schränken wir die Operationen auf $\widehat{A}$ auf monotone (d.h. stetige) Operationen ein:

$$Ops(\widehat{A}) = \bigcup_{(w,s) \in S^* \times S} Ops^{(w,s)}(\widehat{A})$$

mit

$$Ops^{(s_1 \ldots s_n, s)}(\widehat{A}) := [\widehat{A^{s_1}} \times \ldots \times \widehat{A^{s_n}} \rightarrow \widehat{A^s}]$$

(vollständige Halbordnung).

3.4 Rekursive Funktionsschemata

Rekursion hat eine grundlegende Bedeutung für die funktionale Programmierung:

D. Knuth: *I have always felt that the transformation from recursion to iteration is one of the most fundamental concepts of computer science.* (aus: Structured Programming with GOTO Statements, Comp. Surv., 1974)

Zunächst: Signatur wird um Verzweigungssymbole erweitert; das ist wichtig für den Abbruch von Rekursion.

Definition (Signatur mit Verzweigung)
Sei $\Sigma = \langle S, \Omega \rangle$ eine Signatur mit $b \in S$ (*bool*) und $if_s \in \Omega^{(bss,s)}$ für jedes $s \in S$. Dann heißt Σ eine *Signatur mit Verzweigung.*

Definition (Rekusives Funktionsschema)
Sei $\Sigma = \langle S, \Omega \rangle$ eine Signatur mit Verzweigung, X eine S-sortierte Menge von *Argumentvariablen* und V eine $S^* \times S$-sortierte Menge von *Funktionsvariablen.* Dann heißt eine nichtleere Folge R von *Termgleichungen*

$$R = \Big(F_i x_{i1} \ldots x_{in_i} = t_i \Big)_{1 \leq i \leq r}$$

(mit $F_i \neq F_j$ für $i \neq j$) ein *rekursives Funktionsschema* über Σ. (Bezeichnung: $R \in Rek_\Sigma$), falls folgende Typvorschriften erfüllt sind:

- $F_i \in V^{(w_i, s_i)}$,
- $(x_{i1}, \ldots, x_{in_i}) \in X^{w_i}$,
- $t_i \in T^{s_i}_{\Sigma[F_1, \ldots, F_r]}(\{x_{i1}, \ldots, x_{in_i}\})$ mit $\Sigma[F_1, \ldots, F_r] := \langle S, \Omega \cup \{F_1, \ldots, F_r\} \rangle$.

Bezeichnung: Die erste Gleichung ist die *Hauptgleichung*, , wir erklären daher (w_1, s_1) als *Typ* von R.
Schreibweise: $R \in Rek_\Sigma^{(w_1,s_1)}$ und $Rek_\Sigma = \bigcup_{(w,s)\in S^*\times S} Rek_\Sigma^{(w,s)}$.

λ-Notation: Durch λ-Abstraktion läßt sich R notieren als

$$\Big(F_i = \lambda(x_{i1}, ..., x_{in_i}).t_i\Big)_{1\leq i\leq r}$$

(Zur Erinnerung: Terme können in konkreter Syntax verschiedene Darstellungen haben, z.B. „if cond x y“ oder „if cond then x else y“ . Wichtig ist nur die *abstrakte Syntax*; vgl.: Isomorphie der freien Termalgebren)

Beispiel: verschiedene Gofer-Notationen für if
(** Folie 3.2 **)

3.5 Fixpunktsemantik

Wir beschränken uns bei der Interpretation von Σ auf

- flache Halbordnungen mit Wahrheitswerten für b,
- monotone Grundoperationen mit if_s als Verzweigung.

(Dies entspricht im Grunde einer Verallgemeinerung auf partielle Funktionen.)

Definition (Sigma-Interpretation)
Sei $\Sigma = \langle S, \Omega\rangle$ eine Signatur mit Verzweigung und A eine S-sortierte Menge mit $A^b = \{\ true, false\ \}$. Wir erweitern A zu $\widehat{A}$ $(= \bigcup_{s\in S}(A^s \cup \{\perp_s\}))$ und interpretieren Ω durch $\alpha : \Omega \to Ops(\widehat{A})$ (monotone Funktionen!) mit

- $\alpha(if_s)(true, a_1, a_2) := a_1$
- $\alpha(if_s)(false, a_1, a_2) := a_2$
- $\alpha(if_s)(\perp_b, a_1, a_2) := \perp_s$

Dann heißt $\mathcal{A} = \langle\widehat{A}; \alpha\rangle$ eine Σ-*Interpretation.*

Strikte und nicht-strikte Grundoperationen

- $\alpha(if_s) : \widehat{A^b} \times \widehat{A^s} \times \widehat{A^s} \to \widehat{A^s}$ ist stets die übliche nicht-strikte Verzweigung.
- Die übrigen Grundfunktionen werden im allgemeinen als strikt vorausgesetzt (das bedeutet für die Implementierung: normaler Stackcode).
- Boolesche Grundoperationen werden auch nicht-strikt interpretiert, z.B. sequentielles AND, OR oder paralleles AND, OR.

Erweiterung Boolescher Grundoperationen
(** Folie 3.3 **)

Definition (Rekursive Funktionsdefinition erster Ordnung)
Sei $R \in Rek_\Sigma$ und $\mathcal{A}$ eine Σ-Interpretation. Dann heißt $(R, \mathcal{A})$ eine *rekursive Funktionsdefinition erster Ordnung.*

Beispiel: rekursive Funktionsdefinition
(** Folie 3.4 **)

Ziel: Semantiken für „call by name“ - und „call by value“ -Auswertung (lazy/eager evaluation) (nicht-strikte bzw. strikte Semantik)

Voraussetzung

- $\Sigma = \langle S, \Omega \rangle$ Signatur mit Verzweigung
- $\mathcal{A}$ eine *strikte* Σ-Interpretation, d.h. alle Grundoperationen bis auf die Verzweigungen sind strikt: $f_\mathcal{A}(..., \perp^{s_i}, ...) = \perp^s$
  (Grund: leichter zu implementieren; EXEC f $\rightsquigarrow$ Stackcode)
- $R = \Big(F_i = \lambda(x_{i1}, ..., x_{in_i}).t_i\Big)_{1 \leq i \leq r} \in Rek_\Sigma$ mit $\sigma(F_i) = (w_i, s_i)$ für $i = 1, ..., r$

Wir ordnen der rekursiven Funktionsdefinition $(R, \mathcal{A})$ eine stetige Transformation $\Phi_{(R,\mathcal{A})} : FR \rightarrow FR$ des *Funktionenraums*

$$FR := Ops^{(w_1,s_1)}(\widehat{A}) \times ... \times Ops^{(w_r,s_r)}(\widehat{A})$$

zu, so daß für $\overline{g} := (g_1, ..., g_r) \in FR$ gilt: $\overline{g}$ ist genau dann eine Lösung von $(R, \mathcal{A})$, wenn $\overline{g}$ ein Fixpunkt von $\Phi_{(R,\mathcal{A})}$ ist, d.h. $\overline{g} = \Phi_{(R,\mathcal{A})}(\overline{g})$.

- Konstruktion von Φ durch Einsetzen von g_i für F_i in den rechten Gleichungsseiten,
- Φ heißt daher auch *Gleichungs-* oder *Einsetzungsfunktional.*
- Zwei Möglichkeiten bei der Semantik von Variablen und Konstanten: Die Projektionen $\lambda(x_1, ..., x_n).x_i$ und die Konstanten $\lambda(x_1, ..., x_n)$ können strikt bzw. nicht-strikt interpretiert werden.

3.5.1 Nicht-strikte Fixpunktsemantik

Erweiterung der Signatur Σ zu $\Sigma[F_1, ..., F_r] := \langle S, \Omega \cup \{F_1, ..., F_r\} \rangle$ und der Σ-Interpretation $\mathcal{A}$ bzgl. $(g_1, ..., g_r) \in FR$ zu einer $\Sigma[F_1, ..., F_r]$- Interpretation $\mathcal{A}[F_1/g_1, ..., F_r/g_r]$ mit $\alpha(F_i) = g_i$ für $i = 1, ..., r$. Dann ist das *nicht-strikte Einsetzungsfunktional* $\Phi^{ns}_{(R,\mathcal{A})}$ komponentenweise definiert durch

$$\Phi^{ns}_{(R,\mathcal{A})}(g_1, ..., g_r)_i := \mathcal{M}_{\mathcal{A}[F_1/g_1,...,F_r/g_r]}[\lambda(x_{i1}, ..., x_{in_i}).t_i] \qquad \text{für } i = 1, ..., r$$

(Bemerkung: In den t_i stehen die F_j, die in $\mathcal{A}[F_i/g_i]_i$ als g_j interpretiert sind. $\rightarrow$ normale Polynome.)

(Kommentar Indermark: *Metasprachliches Stetigkeitslemma*: Jede Funktion ist stetig, es sei denn, Sie beweisen mir das Gegenteil)

Beachte: In diesem Fall ergeben atomare Terme nicht-strikte Funktionen!

$$\mathcal{M}_{\mathcal{A}[...]}[\lambda(x_{i1},...,x_{in_i}).xij](a_1,...,a_{n_i}) = a_j$$

(auch wenn eines der $a_k = \perp$ ist), und

$$\mathcal{M}_{\mathcal{A}[...]}[\lambda(x_{i1},...,x_{in_i}).c](a_1,...,a_{n_i}) = c_{\mathcal{A}}$$

($\rightsquigarrow$ call-by-name Auswertung)

Lemma: $\Phi^{ns}_{(R,\mathcal{A})} : FR \rightarrow FR$ ist stetig.

Definition (Nicht-strikte Fixpunktsemantik)
Die *nicht-strikte Fixpunktsemantik* der rekursiven Funktionsdefinition $(R, \mathcal{A})$ ist definiert durch

$$\mathcal{M}^{ns}_{\mathcal{A}}[R] := proj_1\Big(fix(\Phi^{ns}_{(R,\mathcal{A})})\Big)$$

(nur erste Komponente, da erste Gleichung Hauptgleichung ist.)

3.5.2 Strikte Fixpunktsemantik

Idee: Projektionen und Konstanten strikt behandeln.
Folgerung: Wenn das Argument nicht definiert ist, ist auch der Funktionswert nicht definiert. (Auswertung: call by value)

Definition (Strikte Termsemantik)
Für $(x_1,...,x_n) \in X^w$ und $t \in T^s_\Sigma(\{x_1,...,x_n\})$ definieren wir bezüglich einer strikten Σ-Interpretation $\mathcal{A}$ und einer Variablenbelegung $\beta : \{x_1,...,x_n\} \rightarrow \widehat{A}$ die *strikte Termsemantik*

$$\mathcal{M}^{str}_{(\mathcal{A},\beta)}[t] \in \widehat{A^s}$$

durch Induktion über die Struktur von t: Sei $\overline{a} = (a_1,...,a_n) \in \widehat{A^{s_1}} \times \widehat{A^{s_n}}$ $(w = s_1...s_n)$ mit $a_i = \beta(x_i)$.

- $t = x_i$: $\mathcal{M}^{str}_{(\mathcal{A},\beta)}[x_i] :=$ $\underline{\text{if}}$ $\overline{a} \in A^w$ $\underline{\text{then}}$ a_i $\underline{\text{else}}$ $\perp^{s_i}$
- $t = c \in \Omega^{(\varepsilon,s)}$: $\mathcal{M}^{str}_{(\mathcal{A},\beta)}[c] :=$ $\underline{\text{if}}$ $\overline{a} \in A^w$ $\underline{\text{then}}$ $c_{\mathcal{A}}$ $\underline{\text{else}}$ $\perp^s$
- $t = ft_1...t_n$: $\mathcal{M}^{str}_{(\mathcal{A},\beta)}[ft_1...t_n] := f_{\mathcal{A}}(\mathcal{M}^{str}_{(\mathcal{A},\beta)}[t_1], ..., \mathcal{M}^{str}_{(\mathcal{A},\beta)}[t_n])$

Die strikte Termsemantik impliziert „strikte“ Polynomfunktionen über $\widehat{A}$: $\mathcal{M}^{str}_{\mathcal{A}}[\lambda(x_1,...,x_n).t] := \left(\widehat{A^{s_1}} \times ... \times \widehat{A^{s_n}} \to \widehat{A^s} : (a_1,...,a_n) \mapsto \mathcal{M}^{str}_{(\mathcal{A},\beta)}[t]\right)$, wobei $\beta(x_i) = a_i$.
Dies führt zum Einsetzungsfunktional $\Phi^{str}_{(R,\mathcal{A})}$ und der *strikten Fixpunktsemantik*:

Definition (Strikte Fixpunktsemantik)
Die *strikte Fixpunktsemantik* der rekursiven Funktionsdefinition $(R,\mathcal{A})$ ist definiert durch

$$\mathcal{M}^{str}_{\mathcal{A}}[R] := proj_1\left(fix(\Phi^{str}_{(R,\mathcal{A})})\right)$$

Folgerung: $\mathcal{M}^{str}_{\mathcal{A}}[R] \leq \mathcal{M}^{ns}_{\mathcal{A}}[R]$ (d.h. die beiden Semantiken unterscheiden sich höchstens im Grad der Definiertheit.)

(** Folie 3.5 ** Nicht-strikte Fixpunktsemantik)
(Beachte: $\lambda x.42$ ist ein Polynom, ein formaler Term, also ein syntaktisches Objekt; $\lambda a.42$ dagegen ist eine Polynomfunktion, eine konkrete Abbildung und damit ein semantisches Objekt.)

(** Folie 3.6 ** Strikte Fixpunktsemantik)
$\mathcal{M}^{str}_{\mathcal{A}}[R] = \lambda a.\bot, \quad \mathcal{M}^{ns}_{\mathcal{A}}[R] = \lambda a.42$

3.6 Reduktionssemantik

Berechnung durch Termersetzung; Vorstufe von Implementierung.

Voraussetzung

- $\Sigma = \langle S, \Omega\rangle$ Signatur mit Verzweigung
- $\mathcal{A}$ eine *strikte* Σ-Interpretation
- $R = \left(F_i = \lambda(x_{i1},...,x_{in_i}).t_i\right)_{1\leq i\leq r} \in Rek_\Sigma$

Definition (Berechnungsterme)
Wir definieren die Menge B der *Berechnungsterme* von $(R,\mathcal{A})$ durch

$$B := T_{\Sigma[F_1,...,F_r]}(A) \qquad (\text{nicht } \widehat{A}!)$$

Beachte: t enthält kein $\bot$ und keine Variablen, wohl aber Funktionsvariablen.

Definition (Reduktionsregeln)

- *Konstantenreduktion*
  $fa_1...a_n \to f_{\mathcal{A}}(a_1,...,a_n)$, falls $f \in \Omega^{(w,s)}, f \neq if_s$ und $(a_1,...,a_n) \in A^w$ (nicht $\widehat{A^w}$!)

- *Verzweigungsreduktion*
  if_s true $u_1 u_2 \to u_1$
  if_s false $u_1 u_2 \to u_2$
  falls $u_1, u_2 \in B^s$.

- *Funktionsaufruf (Kopierregel)*
  $F_i u_1 ... u_{n_i} \to t_i[x_{i1}/u_1, ..., x_{in_i}/u_{n_i}]$
  falls $(u_1, ..., u_{n_i}) \in B^{w_i}$ $(\sigma(F_i) = (w_i, s_i))$

Dabei bezeichnet $t[x_1/u_1, ..., x_n/u_n]$ die Anwendung des durch die Belegung $\beta : \{x_1, ..., x_n\} \to B$ mit $\beta(x_i) = u_i$ bestimmten Einsetzungshomomorphismus

$$\widehat{\beta} : T_{\Sigma[...]}(X) \to T_{\Sigma[...]}(A)$$

auf t, also $t[x_1/u_1, ..., x_n/u_n] = \widehat{\beta}(t)$.

Definition (Reduktionsrelation)
$\Rightarrow \subseteq B^2$ sei durch Induktion über die Termstruktur definiert:

- $u_1 \to u_2$ Reduktionsregel, dann $u_1 \Rightarrow u_2$
- $u \in B$, dann $u \Rightarrow u$
- $\varphi u_1 ... u_n \in B, u_j \Rightarrow v_j$ $(j = 1, ..., n)$, dann $\varphi u_1 ... u_n \Rightarrow \varphi v_1 ... v_n$
  für $\varphi \in \Omega \cup \{F_1, ..., F_r\}$

Satz
$\Rightarrow$ ist konfluent (erfüllt die Church-Rosser-Eigenschaft), d.h. für $u, u_1, u_2 \in B$ mit $u \Rightarrow^* u_1$ und $u \Rightarrow^* u_2$ gibt es ein $u' \in B$ mit $u_1 \Rightarrow^* u'$ und $u_2 \Rightarrow^* u'$.

Korollar
Für $u \in B$ gibt es höchstens ein $a \in A$ mit $u \Rightarrow^* a$.

Bemerkung: Der Nichtdeterminismus der Reduktionsrelation ist ohne semantische Relevanz.

Definition (Reduktionssemantik)
Seien Σ und $\mathcal{A}$ wie oben. Für $R \in Rek_\Sigma^{(w,s)}, (a_1, ..., a_n) \in A^w$ und $a \in A^s$ definieren wir

$$\mathcal{M}_\mathcal{A}^{red}[R](a_1, ..., a_n) = a \qquad :\text{gdw} \qquad F_1 a_1 ... a_n \Rightarrow^* a$$

Satz (Äquivalenz von Reduktions- und nichtstrikter Fixpunktsemantik)
$\mathcal{M}_\mathcal{A}^{ns}[R](a_1, ..., a_n) = a$ gdw $\mathcal{M}_\mathcal{A}^{red}[R](a_1, ..., a_n) = a$.

3.7 Auswertungsstrategien

Ziel: Deterministische Reduktionsrelationen für nichtstrikte und strikte Fixpunkt-Semantik.

3.7.1 Leftmost-Outermost-Strategie

Definition
$\Rightarrow_{LO} \subseteq B^2$ sei definiert durch

- Wenn $u_1 \to u_2$ Reduktionsregel, dann $u_1 \Rightarrow_{LO} u_2$,
- Wenn $fa_1...a_{i-1}u_i...u_n \in B, u_i \notin A, f \in \Omega^{(w,s)}, f \neq if_s$ und $u_i \Rightarrow_{LO} u_i'$, dann $fa_1...a_{i-1}u_i...u_n \Rightarrow_{LO} fa_1...a_{i-1}u_i'u_{i+1}...u_n$,
- Wenn $u \Rightarrow_{LO} u', u, u' \in B^b$, dann $if_s uu_1u_2 \Rightarrow_{LO} if_s u'u_1u_2$.

Bemerkung: Die Leftmost-Outermost-Strategie entspricht dem call-by-name-Funktionsaufruf.

Lemma

1. Die LO-Strategie ist *deterministisch*, d.h. zu jedem $u \in B \setminus A$ gibt es genau ein $u' \in B$ mit $u \Rightarrow_{LO} u'$.
2. Die LO-Strategie ist *korrekt* und *vollständig* bezüglich der nichtstrikten Fixpunktsemantik:

$$\mathcal{M}_{\mathcal{A}}^{ns}[R](a_1, ..., a_n) = a \quad \text{gdw.} \quad F_1a_1...a_n \Rightarrow_{LO}^* a$$

3.7.2 Leftmost-Innermost-Strategie

Definition
$\Rightarrow_{LI} \subseteq B^2$ sei definiert durch

- Wenn $u_1 \to u_2$ kein Funktionsaufruf, dann $u_1 \Rightarrow_{LI} u_2$
(d.h. Konstantenredex oder if true/false)
- $F_ia_1...a_{n_i} \Rightarrow_{LI} t_i[x_{i1}/a_1, ..., x_{in_i}/a_{n_i}]$
- Wenn $\varphi a_1...a_{i-1}u_i...u_n \in B, u_i \notin A, (i > 1 \Rightarrow \varphi = if_s), u_i \Rightarrow_{LI} u_i'$, dann $\varphi a_1...a_{i-1}u_i...u_n \Rightarrow_{LI} \varphi a_1...a_{i-1}u_i'u_{i+1}...u_n$.

Bemerkung: Im letzten Fall gilt entweder $\varphi \in \Omega \setminus \{if_s, s \in S\} \cup \{F_1, ..., F_r\}$ oder ($\varphi \in \{if_s, s \in S\}$ und $i = 1$).

Lemma (Analogon zum Lemma aus 3.7.1)

1. Die LI-Strategie ist *deterministisch*, d.h. zu jedem $u \in B \setminus A$ gibt es genau ein $u' \in B$ mit $u \Rightarrow_{LI} u'$.
2. Die LI-Strategie ist *korrekt* und *vollständig* bezüglich der strikten Fixpunktsemantik:

$$\mathcal{M}_{\mathcal{A}}^{str}[R](a_1, ..., a_n) = a \quad \text{gdw.} \quad F_1a_1...a_n \Rightarrow_{LI}^* a$$

Bemerkung: Für parallele Implementierungen sind Parallel-Outermost- und Parallel-Innermost-Strategien möglich.

(** Folie 3.7 ** Beispielreduktion: call-by-name)
(** Folie 3.8 ** Nachteil der call-by-name-Strategie)
(** Folie 3.9 ** Beispielreduktion: call-by-value)

3.8 Strikter Stackcode

Voraussetzung

- $\Sigma = \langle S, \Omega \rangle$ Signatur mit Verzweigung
- $\mathcal{A} = \langle \widehat{A}; \alpha \rangle$ eine *strikte* Σ-Interpretation
  (strikte Grundfunktionen, nicht-striktes if-then-else)
- $R = \Big(F_i = \lambda(x_{i1}, ..., x_{in_i}).t_i\Big)_{1 \leq i \leq r} \in Rek_\Sigma$

Ziel: Implementierung der Leftmost-Innermost-Strategie auf einer *Stackmaschine*

Idee: Erweiterung der Stackmaschine zur Termauswertung um zwei Komponenten:

- *Befehlszähler BZ*: Verzweigungen und Rekursion erfordern Sprungbefehle.
- *Funktionskeller FK*:
  - Ausführen eines Funktionsaufrufs erfordert das Zwischenspeichern von aktuellen Parametern und Rücksprungadressen in einem *Funktionsblock* (Frame, Aktivierungsblock).
  - Verschachtelte Aufrufstruktur erfordert Speicherung der Funktionsblöcke nach dem Kellerprinzip.
  - FK ersetzt den statischen Hauptspeicher der Stack-Maschine zur Termauswertung.

Stackmaschine mit strikter Auswertung (LI)

(** Bild der Stackmaschine auf Seite 46 **)

Zustandsraum $Z := BZ \times DK \times FK$ mit

- *Befehlszähler* $BZ := I\!N$,
- *Datenkeller* $DK := A^*$ (Spitze rechts) und
- *Funktionskeller* $FK := (A \cup I\!N)^*$ (Spitze links)

Zustand $z = (m, d, h) \in Z$ mit

- *Befehlsmarke* $m \in I\!N$,
- *DK-Zustand* $d = d.p : ... : d.2 : d.1 \in A^*$ und
- *FK-Zustand* $h = h.1 : h.2 : ... : h.q \in (A \cup I\!N)^*$.

Der *Befehlssatz* Cmd, die Menge der Befehle enthält

- *Sprungbefehle* $JMP\ j$ $(j \in I\!N)$ (unbedingt), $JMC\ j$ $(j \in I\!N)$ (bedingt),
- *DK-Befehle* $LOAD\ j$ $(j \in I\!N, j \neq 0)$, $EXEC\ f$ $(f \in \Omega \setminus \{if_s, s \in S\}$,
- *FK-Befehle* $CALL\ (i,n)$ $(i, n \in I\!N)$, RET

und sonst keine.

Programme: $Prog := \{\ 1 : \Gamma_1; ...; k : \Gamma_k \mid \Gamma_i \in Cmd, 1 \leq i \leq k\ \}$
(numerierte Folge von Befehlen)

Befehlssemantik

$$\mathcal{C} : \Gamma \in Cmd \mapsto \mathcal{C}[\![\Gamma]\!] : Z - \to Z$$

$$\begin{array}{ll}
C[\![JMP\ j]\!](m,d,h) & := (j,d,h) \\
C[\![JMC\ j]\!](m,d,h) & := \ \underline{\text{if}}\ d = d' :\ \text{false}\ \ \underline{\text{then}}\ (j, d', h) \\
 & \quad\ \ \underline{\text{else}}\ \ \underline{\text{if}}\ d = d' :\ \text{true}\ \ \underline{\text{then}}\ (m+1, d', h) \\
 & \quad\ \text{(sonst nicht definiert)} \\
C[\![LOAD\ j]\!](m,d,h) & := \ \underline{\text{if}}\ h = ra : pa : a_1 : ... : a_j : h'\ \underline{\text{and}}\ a_j \in A\ \underline{\text{then}}\ (m+1, d : a_j, h) \\
C[\![EXEC\ f]\!](m,d,h) & := \ \underline{\text{if}}\ f \in \Omega^{(w,s)}\ \underline{\text{and}}\ d : d' : \overline{a}\ \underline{\text{and}}\ \overline{a} \in A^w\ \underline{\text{then}}\ (m+1, d' : f_{\mathcal{A}}(\overline{a}), h) \\
C[\![CALL\ (i,n)]\!](m,d,h) & := \ \underline{\text{if}}\ d = d' : \overline{a}\ \underline{\text{and}}\ \overline{a} \in A^n\ \underline{\text{then}}\ (i, d', m+1 : n : \overline{a} : h) \\
C[\![RET]\!](m,d,h) & := \ \underline{\text{if}}\ h = ra : n : a_1 : ... : a_n : h'\ \underline{\text{then}}\ (ra, d, h')
\end{array}$$

Definition (Einzelschrittsemantik)
Sei $= 1 : \Gamma_1; ...; k : \Gamma_k \in Prog$. Die *Einzelschrittsemantik* $\mathcal{E}[\![P]\!] : Z - \to Z$ ist definiert durch

$$\mathcal{E}[\![P]\!](m,d,h) := \ \underline{\text{if}}\ 1 \leq m \leq k\ \underline{\text{then}}\ \mathcal{C}[\![\Gamma_m]\!](m,d,h)$$

Beachte: zwei Möglichkeiten für Nichtdefiniertheit:

1. $m \notin \{1, ..., k\}$,
2. Γ_m nicht ausführbar.

Definition (Iterationssemantik)
$\mathcal{E}$ bestimmt die *Iterationssemantik* $\mathcal{I}[\![P]\!] : Z - \to Z$ mit

$$\mathcal{I}[\![P]\!](m,d,h) := \ \underline{\text{if}}\ m = 0\ \underline{\text{then}}\ (m,d,h)\ \underline{\text{else}}\ \mathcal{I}[\![P]\!]\big(\mathcal{E}[\![P]\!](m,d,h)\big)$$

($\leadsto$ while-Schleifen)

Übersetzung

Übersetzung von rekursiven Funktionsdefinitionen mit strikter Semantik in Stackcode.
Idee: call-by-value Reduktionsstrategie
Beachte: Die Übersetzung ist bereits durch das Funktionsschema bestimmt und von der Interpretation unabhängig!

Definition der Übersetzung durch drei Funktionen:

- die eigentliche Übersetzungsfunktion **s-trans** : $Rek_\Sigma \to Prog$
- die Übersetzung der rechten Gleichungsseiten mit der Funktion *Ausdruckscode (expression translation,* **et***)*
  et : $T_{\Sigma[F_1,\ldots,F_r]}(X) \times I\!N \to Prog'$
  ($I\!N$ für Befehlsadresse)
- die Hilfsfunktion *Codelänge* zur Bestimmung von Befehlsadressen:
  cl : $T_{\Sigma[F_1,\ldots,F_r]}(X) \to I\!N$

wobei $Prog' := \{\ m+1 : \Gamma_1; \ldots; m+k : \Gamma_k \mid m \in I\!N, \Gamma_i \in Cmd, 1 \leq i \leq k\ \}$.

s-trans$\Big(F_i x_{i1} \ldots x_{in_i} = t_i \mid 1 \leq i \leq r\Big) :=$
$\mathbf{et}(t_1, m(F_1))$
$m_1 : RET;$
...
$\mathbf{et}(t_r, m(F_r))$
$m_r : RET$

mit $m(F_1) := 1$, $m_i(F_i) := m(F_i) + \mathbf{cl}(t_i)$, $m(F_{i+1}) := m_i + 1$.

- $\mathbf{et}(x_{ij}, m) := \boxed{m : LOAD\ j;}$
  $\mathbf{cl}(x_{ij}) := 1$
- $\mathbf{et}(f\ u_1 \ldots u_n, m) :=$
  $\mathbf{et}(u_1, m_1)$
  ...
  $\mathbf{et}(u_n, m_n)$
  $m_{n+1} : EXEC\ f;$
  mit $m_1 := m$, $m_{i+1} := m_i + \mathbf{cl}(u_i)$
  $\mathbf{cl}(f\ u_1 \ldots u_n) := 1 + \sum_{i=1}^{n} \mathbf{cl}(u_i)$
- $\mathbf{et}(if_s\ u_1\ u_2\ u_3, m) :=$
  $\mathbf{et}(u_1, m_1)$

$m_1' : JMC\ m_3;$
$\mathbf{et}(u_2, m_2)$
$m_2' : JMP\ m_3';$
$\mathbf{et}(u_3, m_3)$
mit $m_1 := m$, $m_1' := m_1 + \mathbf{cl}(u_1)$,
$m_2 := m_1' + 1$, $m_2' := m_2 + \mathbf{cl}(u_2)$,
$m_3 := m_2' + 1$, $m_3' := m_3 + \mathbf{cl}(u_3)$.
$\mathbf{cl}(if_s\ u_1\ u_2\ u_3) := \sum_{i=1}^{3} \mathbf{cl}(u_i) + 2$

- $\mathbf{et}(F_i\ u_1 ... u_n, m) :=$
  $\mathbf{et}(u_1, m_1)$
  ...
  $m_{n+1} : CALL\ (m(F_i), n)$
  mit $m_1 := m$, $m_{i+1} := m_i + \mathbf{cl}(u_i)$ für $i = 1, ..., n$.
  $\mathbf{cl}(F_i\ u_1 ... u_n) := 1 + \sum_{j=1}^{n} \mathbf{cl}(u_j)$

Satz (Korrektheit und Vollständigkeit der Übersetzung)
Sei $R \in Rek_\Sigma^{(w,s)}$ und $\mathcal{A}$ eine strikte Σ-Interpretation.
Dann gilt für alle $(a_1, ..., a_n, a) \in A^{ws}$:

$$\mathcal{M}_{\mathcal{A}}^{str}[\![R]\!](a_1, ..., a_n) = a \quad \text{gdw.} \quad \mathcal{I}[\![\textbf{s-trans}(R)]\!](1, \varepsilon,\ 0 : n : a_1 : ... : a_n) = (0, a, \varepsilon)$$

3.9 Nicht-strikter Stackcode

Seien $\Sigma, \mathcal{A}$ und R wie in Kapitel 3.8.

Ziel: Implementierung der LO- (Leftmost-Outermost-) Strategie auf einer Stackmaschine

Idee: Auswertung der Argumente eines Funktionsaufrufs $F\ u_1 ... u_n$ erst bei Bedarf $\rightsquigarrow$ *verzögerte Auswertung*, *„lazy evaluation"*

Modifikation der Stackmaschine

Datenkeller $DK := (A \cup I\!N)^*$ ($I\!N$: Sprungmarken)

Funktionskeller $FK := (A \cup I\!N \cup \{X, F\})^*$
Ein *FK-Zustand* $h = h.1 : ... : h.q$ besteht aus einer Folge von Blöcken:

- *F-Blöcke*: $F : ra : pa : \alpha_1 : ... : \alpha_{pa}$
- *X-Blöcke*: $X : ra$

Ein X-Block bestimmt indirekt die für die Argumentberechnung erforderliche Umgebung durch Überspringen des aktuellen F-Blocks.

$\mathbf{env} : FK \times \mathbb{N} - \to FK$
bestimmt bezüglich h uund Schachtelungstiefe l den Rumpf mit aktuellem F-Block an der Spitze.

$$\mathbf{env}(F : h, 0) := F : h$$
$$\mathbf{env}(\overline{F : ra : pa : \alpha_1 : \ldots : \alpha_{pa}} : h, l + 1) := \mathbf{env}(h, l)$$
$$\mathbf{env}(\overline{X : ra} : h, l) := \mathbf{env}(h, l + 1)$$

Zusätzliche Befehle:

- $LOADA\ m$ ($m \in \mathbb{N}, m > 0$, Load Address)
- $EVAL$ (Evaluate)

Befehlssemantik
$JMP\ j$, $JMC\ j$ und $EXEC\ f$ behalten ihre Bedeutung. Modifikation der übrigen Befehle:

$\mathcal{C}[\![LOAD\ j]\!](m, d, h) :=$
$\quad \underline{\text{if}}\ \mathbf{env}(h, 0) = F : ra : pa : \alpha_1 : \ldots : \alpha_{pa} : h'\ \underline{\text{and}}\ j \leq pa\ \underline{\text{and}}\ \alpha_j \in A \cup \mathbb{N}$
$\quad \underline{\text{then}}\ (m + 1, d : \alpha_j, h)$

$\mathcal{C}[\![CALL\ (i, n)]\!](m, d, h) :=$
$\quad \underline{\text{if}}\ d = d' : \overline{\alpha}\ \underline{\text{and}}\ \overline{\alpha} \in (A \cup \mathbb{N})^n$
$\quad \underline{\text{then}}\ (i, d', F : m + 1 : n : \overline{\alpha} : h)$

$\mathcal{C}[\![RET]\!](m, d, h) :=$
$\quad \underline{\text{if}}\ h = X : ra : h'\ \underline{\text{or}}\ h = F : ra : pa : \alpha_1 : \ldots : \alpha_{pa} : h'$
$\quad \underline{\text{then}}\ (ra, d, h')$
(Bei X-Block hochzählen, bei F-Block runterzählen; fertig, falls Zähler auf 0 vor F-Block)

(** Folie 3.13 ** Nichtstrikte Stackmaschine: Variablenzugriff)

Die neuen Befehle:
$\mathcal{C}[\![LOADA\ \widetilde{m}]\!](m, d, h) := (m + 1, d : \widetilde{m}, h)$

$\mathcal{C}[\![EVAL]\!](m, d, h) :=$
$\quad \underline{\text{if}}\ d = d' : \widetilde{m}\ \underline{\text{and}}\ \widetilde{m} \in \mathbb{N}$ (Adresse)
$\quad \underline{\text{then}}\ (\widetilde{m}, d', X : m + 1 : h)$ (X-Block anlegen)
$\quad \underline{\text{else}}\ \ \underline{\text{if}}\ d = d' : a\ \underline{\text{and}}\ a \in A$
$\quad \underline{\text{then}}\ (m + 1, d, h)$ (nichts tun)

Semantik des Stackcodes wie oben.

Übersetzung :

Idee: *call by name*-Reduktionsstrategie
Modifikation der Definition von **s-trans**, **et** und **cl**:

- $\mathbf{et}(x_{ij}, m) :=$
  $m : LOAD\ j;$
  $m+1 : EVAL$
  $\mathbf{cl}(x_{ij} := 2$

- $\mathbf{et}(F_i\ u_1...u_n, m) :=$
  $m_0 : JMP\ m_{n+1};$
  $\mathbf{et}(u_1, m_1)$
  $m_1' : RET;$
  ...
  $\mathbf{et}(u_n, m_n)$
  $m_n' : RET;$
  $m_{n+1} : LOADA\ m_1;$
  ...
  $m_{n+n} : LOADA\ m_n;$
  $m_{2n+1} : CALL\ (m(F_i), n);$
  $\mathbf{cl}(F_i\ u_1...u_n) := \sum_{i=1}^{n} \mathbf{cl}(u_i) + 2n + 2$

Satz (Korrektheit und Vollständigkeit der Übersetzung)
Sei $R \in Rek_{\Sigma}^{(w,s)}$ und $\mathcal{A}$ eine strikte Σ-Interpretation.
Dann gilt für alle $(a_1, ..., a_n, a) \in A^{ws}$:

$$\mathcal{M}_{\mathcal{A}}^{ns}[\![R]\!](a_1, ..., a_n) = a \quad \text{gdw.} \quad \mathcal{I}[\![\mathbf{n\text{-}trans}(R)]\!](1, \varepsilon,\ 0 : n : a_1 : ... : a_n) = (0, a, \varepsilon)$$

(** Folie 3.13 ** Nichtstrikte Stackmaschine: Variablenzugriff)
(** Folie 3.14 ** Beispielübersetzung call-by-name)
(** Folien 3.15-3.17 ** Beispielrechnung call-by-name)
(Statt einem ausgewerteten Argument wird die Adresse ($\rightarrow$ 12) für die Argumentberechnung gespeichert. Der EVAL-Befehl legt einen X-Block an, damit der aktuelle F-Block übersprungen wird; $\leadsto$ einen F-Block tiefer gehen)

Vergleich zwischen strikten und nicht-striktem Stackcode

- strikt: jedes Funktionsargument wird genau einmal berechnet.
  - Vorteil: schnell, platzsparend, falls die Argumente benötigt werden [„Argument benötigt?“ - nicht entscheidbar; entspricht *Halteproblem für Turingmaschinen*. Möglich ist aber eine „Approximation“]
  - Nachteil: Berechnung nicht benötigter Argumente
- nicht-strikt: nur benötigte Argumente werden berechnet, eventuell jedoch mehrfach.

3.10 Call by need

Idee: Kopieren eines ausgewerteten Arguments in den zugehörigen F-Block

Folgerung: Ein Argument wird höchstens einmal berechnet.

Neuer Befehl: $STORE\ j$ $(j \geq 1)$
$\mathcal{C}[\![STORE\ j]\!](m, d, h) :=$
$\underline{\text{if}}$ $d = d' : a$ $\underline{\text{and}}$ $a \in A$ $\underline{\text{and}}$ $h = h.1 : ...h.q$
$\underline{\text{and}}$ $\textbf{env}(h.3 : ... : h.q, 0) = h.p : ... : h.q$
(X-Block mit Länge 2 ausblenden; brauchen F-Block eins höher
$\rightarrow p + 2 + j$ ist Position von α_j $(2 : ra, pa)$)
$\underline{\text{then}}$ $(m + 1, d, h[p + 2 + j/a])$
(Datenkeller nicht ändern; a bleibt)

Änderung der Übersetzung:
$\textbf{et}(F_i\ u_1...u_n, m) :=$
...
$\textbf{et}(u_j, m_j)$
$m'_j : STORE\ j;$
$m'_j + 1 : RET;$
...

Ergebnis: Ein Funktionsargument wird höchstens einmal ausgewertet.

3.11 Endrekursive Funktionsdefinitionen

Spezialfall rekursiver Funktionsdefinitionen: F-Aufruf weder geschachtelt noch innerhalb von Grundfunktionen (mit Ausnahme von Verzweigung)
[$\rightarrow$ das beschreibt genau den Fall iterativer Programme]

Bezeichnung: *iterativ*, *repetitiv*, *tail-recursive*

Folgerung

- Übereinstimmung von strikter und nicht-strikter Semantik
- LO-Strategie und LI-Strategie mit gleichem Ergebnis
- Implementierung ohne Funktionskeller möglich

Implementierung:
FK ersetzen durch *Hauptspeicher* $HS := \{\beta : I\!N - \rightarrow A\}$
Neuer Befehl: $STORE\ n$ $(n \geq 0)$ verschiebt die obersten n DK-Einträge auf die ersten n HS-Zellen.

Übersetzung:
$\textbf{trans}\Big(F_i\ x_{i1}...x_{in_i} = t_i \mid 1 \le i \le r\Big) :=$
$\quad \textbf{et}(t_1, m_1)\ JMP\ 0;$
$\quad ...$
$\quad \textbf{et}(t_r, m_r)\ JMP\ 0;$
mit $\textbf{et}(F_i\ u_1....u_n, m) :=$
$\quad \textbf{et}(u_1, m_1)$
$\quad ...$
$\quad \textbf{et}(u_n, m_n)$
$\quad m' : STORE\ n;$
$\quad m' + 1 : JMP\ m(F_i);$

(** Folie 3.18 ** Beispiel Flußdiagramm)
[$\rightarrow$ Startmarken vor jeder Verzweigung,
$\rightarrow$ *Fortsetzungssemantik*, vergleiche Automaten, Gleichungssystem $L_0 = aL_1 + bL_2$; GTI]

(** Folie 3.19 ** Fortsetzungssemantik zum *Flußdiagramm*)

4 Konstruktoren, unendliche Datenstrukturen

Basiswerte:

a) numerisch, Zahlen, arithmetische Grundfunktionen

b) symbolisch, Terme, Termoperationen (z.B. *Compiler*)

Bemerkung: *Prolog*, *Unifikation*, Σ-Termalgebra

*Konstruktor*en sind Operationssymbole mit *freier* Semantik, also der Termoperation

$$\alpha_T(C)(t_1, ..., t_n) = C\ t_1...t_n$$

Anwendungen:

1. Termfunktionen (z.B. *Compiler*)

2. Datenstrukturen

4.1 Strikte, homogene Konstruktoren

Einzige Datenobjekte sind (Konstruktor-) Terme, keine Zahlen.
Einfachster Fall: *homogene Konstruktoren*, d.h. eine Sorte.

Sei $\Gamma := \bigcup_{n \in \mathbb{N}} \Gamma^{(n)}$ eine Menge (homogener) *Konstruktorsymbol*e. Γ bestimmt als Datenobjekte die Menge T_Γ der *Konstruktorterm*e (Voraussetzung: $\Gamma^{(0)} \neq \emptyset$)

[KOMMENTAR: *vergleiche* $T_\Sigma(X)$; *keine Variablen* $\rightarrow \Gamma^{(0)}$ *nicht leer:* $\exists$ *Konstanten)*]

Zum Rechnen mit Konstruktor-Termen neben Konstruktoren für den Termaufbau zusätzlich *Selektor*en für die Termzerlegung und *Testfunktion*en für „Abfrage auf Termspitze“. Γ induziert auf kanonische Weise eine Signatur mit Verzweigung, die sogenannte *Konstruktorsignatur* $\Sigma_\Gamma := \langle S, \overline{\Gamma} \rangle$ mit

- $S := \{b, c\}$ (boolesche Werte und Konstruktorterme) und
- $\overline{\Gamma} := \Gamma \cup \{ \text{ sel-i } | i \in I\!N, i \geq 1\} \cup \{ \text{ is-C } | C \in \Gamma\} \cup \{\text{if}_s | s \in \{b, c\}\}$
  (Selektoren und Testfunktionen) [KOMMENTAR: *Später: Testen und Selektieren durch* Pattern Matching*; jetzt: explizit*]

*Typvorschrift*en:

- $C \in \Gamma^{(n)} \Rightarrow C \in \overline{\Gamma}^{(c^n, c)}$
- $\text{sel-i} \in \overline{\Gamma}^{(c,c)}$ [nur partiell]
- $\text{is-C} \in \overline{\Gamma}^{(c,b)}$
- $\text{if}_s \in \overline{\Gamma}^{(bss,s)}$

Zu Σ_Γ betrachten wir (zunächst) eine feste strikte Σ_Γ-Interpretation, die *strikte Konstruktor-Interpretation* $\mathcal{A}_\Gamma = \langle \widehat{A}_\Gamma; \alpha_\Gamma \rangle$ mit

- $A^b_\Gamma := I\!B = \{ \text{ false }, \text{ true } \}$
- $A^c_\Gamma := T_\Gamma$

und

- $\alpha_\Gamma(f) := \underline{f}$
- $\underline{C}(t_1, ..., t_n) := C \ t_1...t_n$
- $\underline{\text{sel-i}}(C \ t_1...t_n) := t_i$, falls $1 \leq i \leq n$
- $\underline{\text{sel-i}}(C \ t_1...t_n) := \perp^c$, sonst
  [Grundfunktionen partiell; erweitern]
- $\underline{\text{is-C}}(C \ t_1...t_n) := \text{ true}$
- $\underline{\text{is-C}}(\widetilde{C} \ t_1...t_n) := \text{ false}$ für $\widetilde{C} \neq C$
- if_s wie sonst
  [KOMMENTAR: *später nicht strikt* $\rightsquigarrow$ *unendliche Datenstrukturen mit Hilfe partieller Terme*]

mit strikter Fortsetzung auf die flachen Halbordnungen $\widehat{T}_\Gamma = T_\Gamma \mathbin{\dot\cup} \{\perp^c\}$ und $\widehat{\mathbb{B}} = \mathbb{B} \mathbin{\dot\cup} \{\perp^b\}$.

Folgerung

$$R \in Rek_{\Sigma_\Gamma}^{(c^n,c)}$$

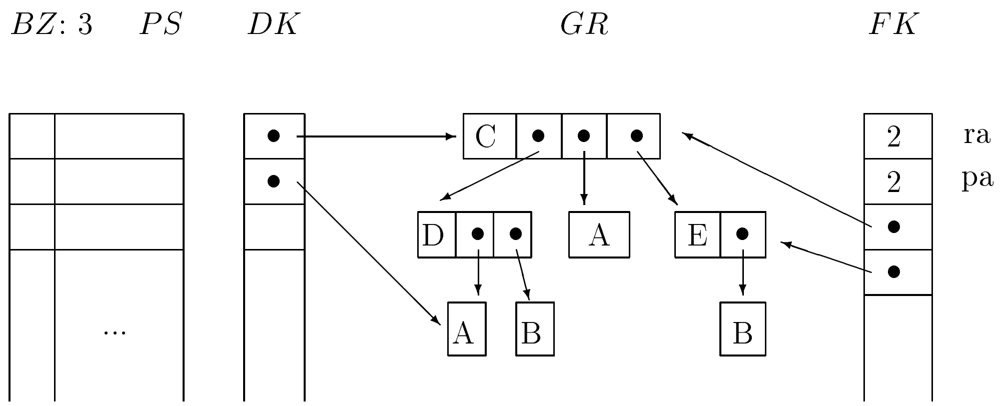
berechnet eine n-stellige Operation auf Konstruktortermen, und zwar eine strikte (cbv) und eine nicht-strikte (cbn) Variante.

Beispiel: $\mathbb{N} : \{0^{(0)}, S^{(1)}\} =: \Gamma \rightsquigarrow \mathbb{N} = T_\Gamma = \{0, S0, SS0, ...\}$
$\to$ *Universalität*, Übung

Modifikation der Stackmaschine (zunächst für den strikten Fall: cbv)

Speicherung eines Terms in einer Zelle ist unrealistisch
$\rightsquigarrow$ Erweiterung der Stackmaschine um eine *Graphkomponente* GR:

Graphmaschine: (vgl. Folie 4.2)



Darstellung eines Konstruktorterms:

Beispiel: $C(D(A), D(B))$ liefert Graph

```
   C
  / \
 D   D
 |   |
 A   B
```

Tabelle:

0	A
1	(D,0)
2	B
3	(D,2) ← Konstruktorknoten
4	(C,1,3)

Modellierung von GR:

- $GAdr := I\!N$ (*Graphadressen*)
- $GKno := \{(C, p_1, ..., p_n) | n \in I\!N, C \in \Gamma^{(n)}, p_i \in GAdr\}$ (*Graphknoten*)
- $GR := \{g | g : GAdr \dashrightarrow GKno, \underline{Def}(g) = \{0, 1, ..., k-1\}, k \in I\!N\}$

Speichervergabe mit der Funktion
$free : GR \to GAdr$,
$free(g) := k$, falls $\underline{Def}(g) = \{0, ..., k-1\}$
durch die Bezeichnung $g[k/(C, p_1, ..., p_n)]$ für die *Grapherweiterung* von g an der Stelle k um den Knoten $(C, p_1, ..., p_n)$
($\to$ wichtig: Speicherfreigabe)

Zustandsraum $Z := BZ \times DK \times FK \times GR$ mit

- $DK := (GAdr \cup I\!B)^*$ *Datenkeller*
- $FK := (GAdr \cup PAdr \cup I\!N)^*$ *Funktionskeller*
  ($Padr$: Programmadressen)

Zustand $z = (m, d, h, g)$

Befehlssatz: Neue Befehle zur Ausführung der strikten Grundoperationen für Konstruktion, Selektion und C-Test anstelle der $EXEC\ f$-Befehle:

- $NODE\ C$ statt $EXEC\ C$
- $TEST\ C$ statt $EXEC$ is-C
- $SEL\ i$ statt $EXEC$ sel-i

(** Folie 4.1 ** Beispiel: Funktionsschema auf Konstruktortermen)

(** Folie 4.2 ** Strikte Graphmaschine)
[KOMMENTAR: „Sharing“: *gemeinsame Zeiger auf Teilterme (*dags*: directed acyclic graphs: keine Zyklen)*
Node C-Befehl legt neuen Konstruktorknoten an]

[KOMMENTAR: *Später: nicht-strikte Graphmaschine*

$\to$ *unendliche Datenstrukturen*

→ *Funktionen höherer Ordnung*

]
Befehlssemantik:

$\mathcal{C}[\![NODE\ C]\!](m,d,h,g) :=$
$\quad\underline{\text{if}}\ C \in \Gamma^{(n)}\ \underline{\text{and}}\ d = d' : p_1 : ... : p_n\ \underline{\text{and}}\ p_1,...,p_n \in GAdr\ \underline{\text{and}}\ free(g) = k$
$\quad\underline{\text{then}}\ (m+1, d' : k, h, g[k/(C, p_1, ..., p_n)])$
[$p_1,...,p_n$ vom DK nehmen und Adresse des neuen Knotens ablegen;
Graph um neuen Knoten C mit Zeigern $p_1,...,p_n$ erweitern]

$\mathcal{C}[\![TEST\ C]\!](m,d,h,g) :=$
$\quad\underline{\text{if}}\ d = d' : p\ \underline{\text{and}}\ g(p) = (C, ...)$
$\quad\underline{\text{then}}\ (m+1, d' :\ \text{true}\ , h, g)$ [p vom DK nehmen]
$\quad\underline{\text{else}}\ \ \underline{\text{if}}\ d = d' : p\ \underline{\text{and}}\ g(p) = (\widetilde{C}, ...)\ \underline{\text{and}}\ \widetilde{C} \neq C$
$\quad\underline{\text{then}}\ (m+1, d' :\ \text{false}\ , h, g)$

[KOMMENTAR: *Instrumentarium für Implementierung des Pattern Matching: TEST und SEL*]

$\mathcal{C}[\![SEL\ i]\!](m,d,h,g) :=$
$\quad\underline{\text{if}}\ d = d' : p\ \underline{\text{and}}\ g(p) = (C, p_1, ..., p_n)\ \underline{\text{and}}\ 1 \leq i \leq n$
$\quad\underline{\text{then}}\ (m+1, d' : p_i, h, g)$
[KOMMENTAR: *beim LOAD i kommt es zu* Sharing ⇝ *wieviele Zeiger zeigen auf ein Objekt?* → *Garbage Collection*]

Übrige Befehle ($JMP\ i, JMC\ i, LOAD\ i, CALL\ (i,n), RET$) eventuell modifizieren.

Programmsemantik:
Übersetzung: $EXEC\ f$ durch neue Befehle ersetzen [KOMMENTAR: *rekursives Funktionsschema übersetzen*]

Anfangszustand der Graphmaschine:
Berechnung von Termfunktionen $f : T_\Gamma^n - \rightarrow T_\Gamma$. *Startcode* zur Darstellung eines Funktionsargumentes $(t_1,...,t_n)$ in der Graphkomponente mit Zeigern auf diese Graphen.

Beispiel: $(t_1,t_2) = (C(A,B),A)$

```
1: NODE A;
2: NODE B;
3: NODE C;
4: NODE A;
5: CALL (m(F1),2);
```

(** Graphen nach einzelnen Befehlen... **)

(** Folien 4.3-4.5 ** Beispiel für Berechnung der Graphmaschine)

Anstelle des Startcodes verwenden wir lieber eine Funktion, die die Eingabe in einen Startzustand umrechnet; also wie bei der „numerischen“ (d.h. nicht-Graph-) Stackmaschine.

4.2 Nicht-strikte, homogene Konstruktoren

Unendliche Listen:

```
list 1 = 1 : list 1
from n = n : from (n+1)
```

in abstrakterer Form:

$F = C(A, F)$

$G\ x = C(x, G(Sx))$

Bisher: C strikt $\Rightarrow F = \perp$ (z.B. über Fixpunktsemantik: $C(A, \perp) = \perp$, da C strikt)

$G = \lambda x. C(x, GSx)$

$= \lambda x.\, \underbrace{C(x, \underbrace{(\lambda x.\perp)(Sx)}_{\perp})}_{\perp,\ \text{da } C \text{ strikt}}$

liefert $G = \lambda x.\perp$

Alternativ: $F = C(A, C(A, C(A, ...\quad)$

```
F:           C
            / \
           A   C
              / \
             A   ...
```

$G\ x = C(x, C(Sx, C(SSx, ...\quad)$

```
G x:         C
            / \
           x   C
              / \
             S   C
             |  / \
             x S   C
               |   ...
               S
               |
               x
```

[KOMMENTAR: *nicht* regulär*: entspricht* $\{a^n b^n | n \in \mathbb{N}\}$*; das Beispiel* F *ist regulär.*]

[KOMMENTAR: *betrachte als* Grammatik*, in der die Nonterminalsymbole Parameter haben.*]

Approximation durch endliche Teilbäume
Nicht-strikte Konstruktoren:
Die flache Halbordnung $\widehat{T_\Gamma} := T_\Gamma \cup \{\perp\}$ reicht nicht aus: $\perp$ tritt innerhalb der Terme auf. Neue Qualität: unendliche Bäume / Terme.
Dies führt zur *vollständigen Halbordnung* CT_Γ (CT = *continuous trees*)

Definition (partieller Γ-Term)
Die Menge

$$PT_\Gamma := T_{\Gamma \cup \perp^{(0)}}$$

der *partiellen* Γ*-Terme* ist halbgeordnet durch die Relation $\leq$ mit

(i) $\perp \leq t$ und $t \leq t$ für alle $t \in PT_\Gamma$

(ii) $C\ t_1 ... t_n \leq C\ t'_1 ... t'_n$, falls $t_i \leq t'_i$ für alle $1 \leq i \leq n$

Beispiel:

```
      C                   C
     / \                 / \
    S   C       <=      S   C
    |  / \              |  / \
   bot  A bot           C  A  S
                       / \    |
                      A   B  bot
```

[$t_1 \leq t_2$ auch auffaßbar als „t_2 *überlagert* t_1]

Darstellung eines unendlichen Terms durch die Menge aller (!) Approximationen aus PT_Γ [eine solche Menge nennen wir ein *Ideal*]
Achtung: PT_Γ ist nicht vollständig, da z.B.

$$\perp \leq S(\perp) \leq S(S(\perp)) \leq ...$$

keine obere Schranke besitzt.

CT_Γ als *Idealvervollständigung* von PT_Γ:

Definition (Ordnungsideal)
Sei $\mathcal{A} = \langle A; \leq \rangle$ eine Halbordnung und $T \subseteq A$ *gerichtet* (d.h.: $\forall a, b \in T\ \exists c \in T : \{a, b\} \leq c$). T heißt *Ideal* (*Ordnungsideal*) $:\Longleftrightarrow$

$$\forall a, b \in A : a \leq b, b \in T \Rightarrow a \in T$$

Beispiel: *Hauptideal*
Für jedes $a \in A$ bestimmt $Id(a) := \{b \in A | b \leq a\}$ das sogenannte *Hauptideal* von a.

Satz
Sei $\mathcal{A} = \langle A; \leq \rangle$ eine Halbordnung mit kleinstem Element $\perp_{\mathcal{A}}$. Dann bildet

$$Id(\mathcal{A}) := \{T \subseteq A | T \text{ Ideal von } \mathcal{A}\}$$

mit $T_1 \leq T_2 :\Longleftrightarrow T_1 \subseteq T_2$ eine vollständige Halbordnung.

Beweis: Sei $\mathcal{T} \subseteq Id(\mathcal{A})$ gerichtet. $\bigsqcup \mathcal{T} := \bigcup\{T | T \in \mathcal{T}\}$ ist ein Ideal:
$a, b \in \bigsqcup \mathcal{T} \Rightarrow \exists T_1, T_2 \in \mathcal{T} : a \in T_1, b \in T_2$. Da $\mathcal{T}$ gerichtet ist, gibt es eine gemeinsame Obermenge $T_0 \in \mathcal{T}$: $T_1, T_2 \subseteq T_0$. Da T_0 (als Ideal) gerichtet ist, gibt es ein $c \in T_0$ mit $a, b \leq c$.
Teilmengeneigenschaft von $\bigsqcup \mathcal{T}$ klar.

Folgerung
$\langle \mathcal{A}; \leq \rangle$ ist durch $a \mapsto Id(a)$ in $\langle Id(\mathcal{A}); \leq \rangle$ einbettbar.

Anwendung auf Konstruktorterme:

$$CT_\Gamma := Id(\langle PT_\Gamma; \leq \rangle)$$

CT_Γ bildet bezüglich Inklusion die vollständige Halbordnung der *unendlichen* Γ-*Terme*. Sie enthält über die Einbettung $t \mapsto Id(t)$ die Halbordnung der partiellen Terme PT_Γ.
(Zur Bezeichnung: CT = *complete trees*)

Zerlegungslemma
Sei $T \in CT_\Gamma$ und $T \neq \{\perp\}$. Dann existieren $C \in \Gamma^{(n)}$ und $T_1, ..., T_n \in CT_\Gamma$ mit

$$T = \{C\ t_1 ... t_n \mid t_i \in T_i, 1 \leq i \leq n\} \cup \{\perp\} =: C\ T_1 ... T_n$$

Dabei sind C, $T_1, ..., T_n$ eindeutig bestimmt.

Beweis: Übung.

Beispiel: $\{C(A, \perp), C(A, C(A, \perp)), ...\} \cup \{\perp\}$ ist gerichtet (aber noch kein Ideal! z.B. $A \rightsquigarrow \perp$).

Totale Γ-Terme: T_Γ sei die Menge der partiellen Bäume ohne $\perp$-Blätter. Dann gilt

$$T_\Gamma \subseteq PT_\Gamma \subseteq CT_\Gamma$$

wobei die zweite Inklusion im Sinne der Einbettung aufzufassen ist.

Nicht-strikte Konstruktoren
Σ_Γ bestimmt die folgende *nicht-strikte Konstruktor-Interpretation* mit CT_Γ als Wertebereich (neben $\widetilde{\mathbb{B}}$):

- $C \in \Gamma^{(n)} \mapsto \underline{C}(T_1, ..., T_n) := C\ T_1...T_n$

[KOMMENTAR: *1976:* „Cons should not evaluate its arguments“*; wurde in LISP-Implementierungen nicht hinreichend berücksichtigt. (*Friedman, Wyse*)*]

- $\underline{\text{sel-i}}(C\ T_1...T_n) := T_i$, falls $1 \leq i \leq n$
  $\underline{\text{sel-i}}(C\ T_1...T_n) := \{\perp\}$, sonst
  $\underline{\text{sel-i}}(\{\perp\}) := \{\perp\}$

- $\underline{\text{is-C}}(C\ T_1...T_n) :=$ true
  $\underline{\text{is-C}}(\widetilde{C}\ T_1...T_n) :=$ false für $\widetilde{C} \neq C$
  $\underline{\text{is-C}}(\{\perp\}) := \perp^{bool}$

Bemerkung zu $\underline{\text{is-C}}$: $CT_\Gamma \to \widetilde{I\!B}$. Würde man $\underline{\text{is-C}}(\{\perp\}) =$ false definieren, so wäre $\underline{\text{is-C}}$ unstetig ($\to$ Fixpunktsatz nicht anwendbar), denn:
$\{\perp\} \leq \{C...\}$, aber für die Bilder gilt false $\not\leq$ true , also: nicht monoton $\Rightarrow$ nicht stetig.

Verschiedene Grade der Nicht-Striktheit:
Konstruktoren: $\underline{C}(\{\perp\}, T_2, ..., T_n) := C\ \{\perp\}T_2...T_n$ (*starke Nicht-Striktheit*)
Selektion / Test: $\underline{\text{sel-2}}(C\ \{\perp\}T_2) = T_2$, $\underline{\text{is-}C}(C\ \{\perp\}\{\perp\}) =$ true (*schwache Nicht-Striktheit*)

Beispiel: Wir suchen Lösungen der Gleichung $F(x) = C(x, F(x))$:

a) strikt: $T_\Gamma \cup \{\perp\} \rightsquigarrow F = \lambda x.\perp$ (flache Halbordnung)

b) nicht-strikt: CT_Γ nicht-strikte Konstruktorinterpretation

Fixpunktsemantik mit nicht-strikten Konstruktoren

Definition (nicht-strikte Fixpunktsemantik)
Sei $R \in Rek^{(c^n,c)}_{\Sigma_\Gamma}$, Interpretation CT_Γ. (R, CT_Γ) bestimmt eine stetige Transformation

$$\Phi_{(R,CT_\Gamma)} : FR \to FR \quad \textit{Einsetzungsfunktional}$$

(alles ist nicht-strikt, insbesondere auch Projektionen und konstante Funktionen)
Die *nicht-strikte Fixpunktsemantik* ist definiert durch

$$\mathcal{M}_{CT_\Gamma}[\![R]\!] := proj_1(fix(\Phi_{(R,CT_\Gamma)})) : CT^n_\Gamma \to CT_\Gamma$$

Definition (Reduktionssemantik)
Kein Rechnen auf unendlichen Termen, nur potentiell unendliche Terme, Datenstrukturen
[KOMMENTAR: *interessanter: heterogene Konstruktoren, z.B. cons* $: X \times [X] \to [X]$]
Hier: nur endlicher Anteil der Fixpunktsemantik:

$$M^{fin}_{CT_\Gamma}[\![R]\!] : T^n_\Gamma - \to T_\Gamma$$

Definition (Berechnungsterm)
$B := T_{\Sigma_\Gamma[F_1,...,F_r]}(\emptyset) = T_{\Sigma_\Gamma[F_1,...,F_r]}$
Die Berechnungsterme enthalten: C, is-C, sel-i, if, F; nicht: $x, \perp$.
$B = B^b \cup B^c$.

irreduzible Normalformen:

- { true , false } $\subseteq B^b$
- $T_\Gamma \subseteq B^c$

[KOMMENTAR: *Bei Selektion muß ein Term solange ausgewertet werden, bis das Kopfsymbol (Konstruktor!) feststeht. Daher:*]

Definition (Kopfnormalform)
$C\ t_1...t_n \in B^c$ heißt in *Kopfnormalform.*

Reduktionsregeln:

- *Konstantenreduktion*:
  sel-i $C\ t_1...t_n \to t_i$, falls $C \in \Gamma^{(n)}, 1 \leq i \leq n$ und $t_1, ..., t_n \in B^c$
  is-C $C\ t_1...t_n \to$ true, falls $C \in \Gamma^{(n)}$ und $t_1, ..., t_n \in B^c$
  is-C $\widetilde{C}\ t_1...t_n \to$ false, falls $C \in \Gamma^{(n)}$, $t_1, ..., t_n \in B^c$ und $C \neq \widetilde{C}$.
  [KOMMENTAR: *Argument von sel-i / is-C muß in Kopfnormalform sein. Steht vorne ein F, dann kann nicht reduziert werden.*]
- *Verzweigungsreduktion* (vie vorher)
- *Funktionsaufruf* (wie vorher)

Reduktionssemantik: wie früher (nicht-deterministisch und konfluent)

Korollar: Für $t \in B$ gibt es höchstens eine irreduzible Normalform γ mit $t \Rightarrow^* \gamma$.

Definition (Reduktionssemantik)
Für $R \in Rek_{\Sigma_\Gamma}^{(c^n,c)}$, $\gamma_1, ..., \gamma_n, \gamma \in T_\Gamma$ definieren wir

$$\mathcal{M}_{CT_\Gamma}^{red}[\![R]\!] \ :\Longleftrightarrow \quad F_1\gamma_1\ ...\gamma_n \Rightarrow^* \gamma$$

Satz: $\mathcal{M}_{CT_\Gamma}^{fin}[\![R]\!] = \mathcal{M}_{CT_\Gamma}^{red}[\![R]\!]$
[KOMMENTAR: *Beweis: Diplomarbeit Schitil*]

Auswertungsstrategie:
LO-Strategie: erster reduzierbarer Teilterm von $t \in B$ in Präfixnotation

- Falls $t_1 \to t_2$, dann $t_1 \Rightarrow_{LO} t_2$.
- $C\ \gamma_1...\gamma_{i-1}t_i...t_n \Rightarrow_{LO} C\ \gamma_1...\gamma_{i-1}t_i't_{i+1}...t_n$,
  falls $C \in \Gamma^{(n)}, \gamma_1, ..., \gamma_{i-1} \in T_\Gamma$ (irreduzibel) und $t_i \Rightarrow_{LO} t_i'$.

- is-C $t \Rightarrow_{LO}$ is-C t', falls $t \Rightarrow_{LO} t'$ und t nicht in Kopfnormalform
- sel-i $t \Rightarrow_{LO}$ sel-i t', falls $t \Rightarrow_{LO} t'$ und t nicht in Kopfnormalform
- if t_0 t_1 $t_2 \Rightarrow_{LO}$ if t'_0 t_1 t_2, falls $t_0 \Rightarrow_{LO} t'_0$
  (d.h. $t_0 \neq$ true,false, denn z.B. true $\not\Rightarrow_{LO}$)

Lemma: Die LO-Strategie ist deterministisch sowie korrekt und vollständig bezüglich $\mathcal{M}^{fin}$.

(** Folie 4.6 ** Beispielreduktion LO)
(** Folie 4.7 ** Beispiel: $+(x, y)$ mit *Suspensionsknoten*)

Zur Implementierung:
Idee: Auswertung von Argumenten verzögern
Bisher: Kellertechnik mit Zugriff auf tiefere Funktionsblöcke ausreichend („*Deletion-Strategie*") (d.h.: nach Funktionsaufruf werden Daten gelöscht)
Jetzt: Ergebnis einer Funktion kann wegen der verzögerten Berechnung von Konstruktortermen abhängig von Parametern sein („*Retention-Strategie*")

Beispiel: $F\ x$ =sel-1$(G\ x)$, $G\ x = Cons(x + 1, Nil)$
Aufruf: $F(17)$
Im ersten Schritt wird ein Block für F aufgebaut: [0|1|17].
Um das Argument von sel zu verarbeiten, wird G aufgerufen: [$\bullet$|1|17||0|1|17], [$Cons|\alpha|$].
Dabei ist α die Adresse für den Code von $x + 1$ - dieser benötigt die 17 aus dem G-Block! Bei Kellertechnik würde aber der G-Block zu früh gelöscht.
Stattdessen erzeugen wir einen *Suspensionsknoten*:
$[Cons|\bullet \downarrow |]$
[SUSP|α|1|17] ($1 = n$: Anzahl der Parameter)

Lösung: Suspensionsknoten als Argumente von Konstruktorknoten, mit Codeadresse und Parametern für spätere Argumentberechnung (= retention strategy)
(SUSP,$ca, n, p_1, ..., p_n$): Closure / Abschluß

Modifikation der Graphmaschine:

- Weitere Graphknoten: (SUSP,$ca, n, p_1, ..., p_n$)
  mit $ca \in PAdr, n \in \mathbb{N}, p_i \in GAdr$
- Zusätzliche Befehle:
  SUSPEND ca (Erzeugung von Suspensionsknoten)
  EVAL (Suspensionsknoten auflösen, falls er gebraucht wird (Auswertung)

Befehlssemantik:
$\mathcal{C}[\![SUSPEND\ ca]\!](m, d, h, g) :=$
  $\underline{\text{if}}\ ca \in PAdr\ \underline{\text{and}}\ h = ra : n : p_1 : ... : p_n : h'\ \underline{\text{and}}\ free(g) = k$
  $\underline{\text{then}}\ (m + 1, d : k, h, g[k/(SUSP, ca, n, p_1, ..., p_n)])$

[KOMMENTAR: *h, nicht h': nur Kopieren des obersten Blocks*]

$\mathcal{C}[\![EVAL]\!](m, d, h, g) :=$
$\underline{\text{if}}\ d = d' : p\ \underline{\text{and}}\ g(p) = (C, ...)$
$\underline{\text{then}}\ (m+1, d, h, g) \qquad [NOP]$
$\underline{\text{else}}\ \ \underline{\text{if}}\ d = d' : p\ \underline{\text{and}}\ g(p) = (SUSP, ca, n, p_1, ..., p_n)$
$\underline{\text{then}}\ (ca, d', m+1 : n : p_1 : ... : p_n : h, g)$

[KOMMENTAR: *Am Ende der Berechnung soll auf dem Datenkeller statt eines Zeigers auf einen Suspensionsknoten ein Zeiger auf einen Konstruktorknoten stehen.*]

Übersetzung: ähnlich zu nicht-striktem Stackcode, jedoch hier keine X-Blöcke sondern *SUSP-Knoten.*

Modifikation von **et**:

- $\mathbf{et}(x_{ij}, m) :=$
  $m : LOAD\ j;$
  $m+1 : EVAL;$
- $\mathbf{et}(\text{sel-i}u, m) :=$
  $\mathbf{et}(u, m)$
  $m_1 : SEL\ i;$
  $m_1 + 1 : EVAL;$
- $\mathbf{et}(\text{is-}Cu, m) :=$
  $\mathbf{et}(u, m)$
  $m_1 : TEST\ C;$
- $\mathbf{et}(if\ \ u_0u_1u_2, m)$: wie bisher
- $\mathbf{et}(C\ \ u_1...u_n, m) :=$
  $m : JMP\ m';$
  $\mathbf{et}(u_1, m_1)$
  $m_1' : RET;$
  ...
  $\mathbf{et}(u_n, m_n)$
  $m_n' : RET;$
  $m' : SUSPEND\ m_1;$
  ...
  $m' + n - 1 : SUSPEND\ m_n;$
  $NODE\ C;$
- $\mathbf{et}(F_i\ \ u_1...u_n, m) :=$
  $m : JMP\ m'$
  $\mathbf{et}(u_1, m_1)$
  $m_1' : RET;$

...
$\mathbf{et}(u_n, m_n)$
$m'_n : RET;$
$m' : SUSPEND\ m_1;$
...
$m' + n - 1 : SUSPEND\ m_n;$
$CALL\ (m(F_i), n);$

Dieser nicht-strikte Graphcode berechnet die Funktion

$$\mathcal{M}^{fin}_{CT_\Gamma}[\![R]\!] : T^n_\Gamma \;-\!\!\rightarrow T_\Gamma$$

nur im folgenden Sinn:

Eingabe-Kodierung:
$\underline{\text{input}}\ (t_1, ..., t_n) = (1, \varepsilon, 0 : n : p_1 : ... : p_n, g)$
mit $g(p_i)$:=die Graphdarstellung von $t_i \in T_\Gamma$
Das Ergebnis kann noch Suspensionsknoten enthalten. Daher:
$\underline{\text{output}}\ (0, p, \varepsilon, g) := \underline{se}(g, p)$ (se = Suspensionsknoten-Eliminierung) und
$\underline{se} : GR \times GAdr \;-\!\!\rightarrow T_\Gamma$.

(** Folie 4.7 ** Beispiel: $+(x, y)$ mit Suspensionsknoten

4.3 Heterogene Konstruktoren

Konstruktoren zur Modellierung von Datenstrukturen (Listen, Bäume, ...)
Auftreten verschiedener Typen

Bisher: Zwei Welten: a) numerisch, b) symbolisch

a) $\Sigma = \langle S, \Omega \rangle$ Signatur mit Verzweigung,
strikte Grundoperationen,
strikte oder nicht-strikte Projektionen und konstante Funktionen;
flache Halbordnung, kein Heap

b) Σ_Γ Konstruktorsignatur,
nicht-strikte Grundoperationen,
unendliche Bäume (Terme);
Heap mit Suspensionsknoten

Jetzt: Beide Welten vereinigen: numerisch und symbolisch

Standardbeispiel: Listen: int, list
$cons : int \times list \rightarrow list,$
$nil : \ \rightarrow list$

Definition (Konstruktorbasierte Signatur mit Verzweigung)
Sei $\Sigma = \langle S, \Omega\rangle$ eine Signatur mit Verzweigung. Σ heißt *konstruktorbasiert* $:\Longleftrightarrow$

- $S = BS \,\dot\cup\, CS$ (Basis- und Konstruktorsorten)
- $\Omega = BOp \,\dot\cup\, COp \,\dot\cup\, TOp \,\dot\cup\, SOp$ (Grund-, Konstruktor-, Test- und Selektionsoperationen)
- BOp ist eine $BS^* \times BS$-sortierte Menge (der Grundoperationssymbole)
- COp ist eine $S^* \times CS$-sortierte Menge (Konstruktoren)
  [KOMMENTAR: *Argumentsorten sowohl Basis- als auch Konstruktorsorten*]
- $TOp := \{$ is-$C \mid C \in COp, \sigma(C) = (s_1...s_n, s), 1 \leq j \leq n\}$
  $\sigma(C) = (s_1...s_n, s) \Rightarrow \sigma(\text{sel-}C\text{-}j = (s, s_j)$.

Definition (Konstruktorbasierte Interpretation)
Sei $\Sigma = \langle S, \Omega\rangle$ eine konstruktorbasierte Signatur mit Verzweigung. Sei $\mathcal{A} = \langle \widehat{A}; \alpha\rangle$ eine $\langle BS, BOp\rangle$-Interpretation [Striktheit!]. Diese bestimmt eine *Konstruktorbasierte Σ-Interpretation*

$$C\mathcal{A} := \langle \widehat{A} \,\dot\cup\, \bigcup\{CT^s_{COp} | s \in CS\}; \quad \gamma\rangle$$

(Beachte: an den Blättern der COp-Bäume sind $\widehat{A}$-Elemente möglich.)

- $\gamma(f) := \alpha(f)$ für $f \in BOp$
- $\gamma(C)(t_1, ..., t_n) := C\ t_1...t_n$
  (unendliche Bäume; schreibe wieder t_i statt T_i)
- $\gamma(\text{is-}C)(t)$:=true, falls $t = C...$
  $\gamma(\text{is-}C)(t)$:=false, falls $t = \widetilde{C}..., \widetilde{C} \neq C$
- $\gamma(\text{sel-}C\text{-}j)(C\ t_1...t_n) := t_j$
  („sonst“ $:= \perp^{s_j}$)

Implementierung:
Zusätzliche Befehle

- BOX: verwandelt $a \in A$ vom DK in einen Heapknoten.
- $UNBOX$: holt den Inhalt eines Datenknotens auf den DK.

Die Maschine besitzt dann drei Arten von Knoten:

- *Konstruktorknoten*
- *Suspensionsknoten*
- *Datenknoten*

(** Folie 4.8 ** Beispiel: heterogene Konstruktoren)
(** Folie 4.9 ** Vereinfachte Programmierung mit *CASE*-Befehl)

4.4 Pattern Matching

Einfachere Programmierung durch implizites Testen und Selektieren von Konstruktortermen beim *Pattern Matching*.
Syntaktische Alternative: Argumentmuster auf linken Seiten von Funktionsdefinitionen

Beispiel: $F(0) = S(0)$, $F(S(n)) = S(n) * F(n)$ statt
$F(n) =$ $\underline{\text{if}}$ is-0(n) $\underline{\text{then}}$ $S(0)$ $\underline{\text{else}}$ $n * F(\text{sel-1}(n))$

Allgemeiner:

$$F : CT_\Gamma - \to CT_\Gamma; \quad \Gamma = \{C_1, ..., C_r\}$$

eine Funktion F wird beschrieben durch
$F(C_1\ x_{11}...x_{1n_1}) = t_1$
...
$F(C_r\ x_{r1}...x_{rn_r}) = t_r$
mit $t_i \in T_\Gamma(x_{i1}, ..., x_{in_i})$.

Bemerkungen
Vorteile:

- keine explizite Verzweigung, Test, Selektion erforderlich
- übersichtlicher

Allgemeinere Pattern sind möglich.

Implementierung:

a) Übersetzung in Test/Selektor-Form

b) Verwendung eines neuen *CASE*-Befehls
$CASE\ (C_1, \alpha_1, ..., C_r, \alpha_r)$
$\Gamma = \{C_1, ..., C_r\}, \alpha_i \in PAdr$

(** Folie 4.9 ** Beispiel: Übersetzung mit CASE)

5 Funktionen höherer Ordnung

Bisher: nur Basiswerte als Argumente von Funktionen zugelassen; rekursive Funktionsdefinitionen erster Ordnung
Jetzt: auch Funktionen als Argumente und Werte zugelassen → *Funktionale* bzw. *Funktionen höherer Ordnung*

Beispiel:
$fold :: (a \to a \to a) \to a \to ([a] \to a)$

$fold\ f\ c\ [] = c$
$fold\ f\ c\ (h:t) = h\ `f`\ (fold\ f\ c\ t)$
Damit lassen sich z.B. *sum* und *prod* ausdrücken:

- $sum\ i = fold\ (+)\ 0\ [1..i]$
- $prod\ i = fold\ (*)\ 1\ [1..i]$

Skelett: Abstraktion algorithmischer Muster, z.B. DC (divide/conquer)
HOF's (higher order functions): mächtige Programmiertechnik von FPS (Funktionalen Programmier-Sprachen)

Formaler Rahmen: *Lambda-Kalkül* (*Church*, 1936)
Konstruktion von Funktionen durch *Abstraktion* und *Applikation*:

- *Abstraktion*: $5 \rightsquigarrow \lambda x.5$
  $F(x,y) = t \rightsquigarrow F = \lambda(x,y).t$
  ($\approx$ Lifting vom Datenlevel auf Funktionslevel)
- *Applikation*: $\big(\lambda(x,y).t(5,3)\big)$
  $\rightsquigarrow$ (β-Reduktion) $t[x/5, y/3]$

Zwei Arten von Lambda-Kalkül:

- *ungetypter Lambda-Kalkül*:
  Selbstapplikation (EE)
  universell: jede berechenbare Funktion ist λ-definierbar; mit Selbstapplikation ist Rekursion simulierbar.
- *getypter Lambda-Kalkül*:
  getypte Applikation
  erfordert explizite Rekursion

Currying: Der kanonische *Curry*-Isomorphismus

$$I : [A \times B \to C] \to [A \to [B \to C]]; \quad I(f)(a)(b) := f(a,b)$$

(keine Stetigkeit im Symbol [... → ...]) erlaubt die Elimination kartesischer Typen; statt $\sigma(f) = s_1 \times ... \times s_n \to s$ einfach $s_1 \to ... \to s_n \to s$
Konvention: Klammerung rechtsassoziativ

partielle Applikation: fa, fab (es müssen nicht „alle" Argumente übergeben werden.)

5.1 Der getypte $\lambda\mu$-Kalkül

Der getypte $\lambda\mu$-Kalkül ist ein getypter Lambda-Kalkül mit Rekursion und Funktionen höherer Ordnung (ohne kartesische Typen)

Syntax: Sei $\Sigma = \langle S, \Omega \rangle$ eine Signatur mit *bool* $\in S$ [if kein Grundsymbol!]

Definition (Typ höherer Ordnung)
Die Menge $Typ(S)$ der *Typen höherer Ordnung* über S ist induktiv definiert durch

- $S \subseteq Typ(S)$
- $t_1, t_2 \in Typ(S) \Rightarrow (t_1 \to t_2) \in Typ(S)$

Konvention: Klammern können weggelassen werden, wenn man Rechtsklammerung vereinbart: $t_1 \to t_2 \to t_3 = (t_1 \to (t_2 \to t_3))$

Definition ($\lambda\mu$-Ausdruck über Σ)
Sei V eine $Typ(S)$-sortierte Menge von Variablen, so daß für jeden Typ $t \in Typ(S)$ V^t abzählbar ist.
Dann definieren wir die Menge $Exp(\Sigma)$ der $Typ(S)$-sortierten *$\lambda\mu$-Ausdrücke* über Σ als kleinste Menge E mit folgenden Eigenschaften:

1. $\Omega^{(\varepsilon,s)} \subseteq E^s$, $\Omega^{(s_1 \ldots s_n, s)} \subseteq E^{s_1 \to \ldots \to s_n \to s}$ (Konstanten)
2. $V \subseteq E$ (Variablen)
3. $e \in E^{bool}, e_1, e_2 \in E^t \Rightarrow$ $\underline{\text{if}}$ e $\underline{\text{then}}$ e_1 $\underline{\text{else}}$ e_2 $\underline{\text{fi}}$ $\in E^t$ (Verzweigung)
4. $x \in V^{t_1}, e \in E^{t_2} \Rightarrow \lambda x.e \in E^{t_1 \to t_2}$ (Abstraktion)
5. $e_1 \in E^{t_1 \to t_2}, e_2 \in E^{t_1} \Rightarrow (e_1 e_2) \in E^{t_2}$ (Applikation)
6. $x \in V^t, e \in E^t \Rightarrow \mu x.e \in E^t$ (Rekursion)

Beispiel: *Fakultätsfunktion*
$\mu F.\lambda x.$ $\underline{\text{if}}$ $((=x)0)$ $\underline{\text{then}}$ 1 $\underline{\text{else}}$ $((*x)F((-x)1)))$ $\underline{\text{fi}}$

[KOMMENTAR: *vgl. Satz von* Kleene *über reguläre Ausdrücke / endliche Automaten Analog:*

- *einstellige Rekursion (universell) (skalare Rekursion, aber geschachtelt)*
- *Kombinatorsysteme (Gleichungen) (nur eine äußere Rekursion)*

Vgl. Gleichungssysteme zur Bestimmung der von einem endlichen Automaten akzeptierten Sprache]

(** Folie 5.1 ** $\lambda\mu$-Ausdrücke)

Fixpunktsemantik:

Definition (Funktionenraum)
Sei $\mathcal{A} = \langle \widehat{A}; \alpha \rangle$ eine strikte Σ-Interpretation. ($\underline{\text{if}} \notin \Sigma$) Diese bestimmt die $Typ(S)$-sortierte Menge $Fun(\mathcal{A})$ der *Funktionenräume über* $\mathcal{A}$:

$$Fun(\mathcal{A})^s := \widehat{A^s} = A^s \cup \{\perp^s\}; \quad Fun(\mathcal{A})^{t_1 \to t_2} := [Fun(\mathcal{A})^{t_1} \to Fun(\mathcal{A})^{t_2}]$$

[stetige Funktionen!]

Wir wollen freie und gebundene Variablen unterscheiden können:

Definition (Umgebung)
$Env(\mathcal{A})$ sei die Menge der *Umgebungen über* $\mathcal{A}$:

$$\rho \in Env(\mathcal{A}) :\Longleftrightarrow \rho : V \to Fun(\mathcal{A}) \text{ sortentreu}$$

Eine Umgebung gibt Werte für die freien Variablen vor.

Definition (Ausdruckssemantik)

$$\mathcal{E} : Exp(\Sigma) \times Env(\mathcal{A}) \to Fun(\mathcal{A})$$

wird induktiv über den Aufbau von $Exp(\Sigma)$ festgelegt: $\mathcal{E}[\![e]\!]\rho \in Fun(\mathcal{A})$:

1. $\mathcal{E}[\![c]\!]\rho := c_{\mathcal{A}}\ (= \alpha(c))$
   $\mathcal{E}[\![f]\!]\rho := f_{\mathcal{A}}$, eigentlich $I(f_{\mathcal{A}})$, *Curry*
2. $\mathcal{E}[\![x]\!]\rho := \rho(x), x \in V$
3. $\mathcal{E}[\![\ \underline{\text{if}}\ e\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2\ \underline{\text{fi}}\ ]\!]\rho :=$
   $\mathcal{E}[\![e_1]\!]\rho$, falls $\mathcal{E}[\![e]\!]\rho =$ true
   $\mathcal{E}[\![e_2]\!]\rho$, falls $\mathcal{E}[\![e]\!]\rho =$ false
   $\perp$, falls $\mathcal{E}[\![e]\!]\rho = \perp$
4. $\mathcal{E}[\![\lambda x^t.e]\!]\rho := \Big(Fun(\mathcal{A})^t \to Fun(\mathcal{A}) : v \mapsto \mathcal{E}[\![e]\!]\rho[x/v]\Big)$
5. $\mathcal{E}[\![(ee_1)]\!]\rho := \mathcal{E}[\![e]\!]\rho\Big(\mathcal{E}[\![e_1]\!]\rho\Big)$
6. $\mathcal{E}[\![\mu x^t.e]\!]\rho := fix\Big(v \in Fun(\mathcal{A})^t \mapsto \mathcal{E}[\![e]\!]\rho[x/v]\Big)$
   („*Einsetzungsfunktional*“)

(** Folie 5.2 ** Freie / gebundene Variablen)

$free(t), bound(t)$. Variablen können in einem Term sowohl frei als auch gebunden sein: $\exists t : free(t) \cap bound(t) \neq \emptyset$

Definition (Programm)

1. $e \in Exp(\Sigma)$ mit $free(e) = \emptyset$ heißt *geschlossen*.

2. Sei $w = s_1...s_n \in S^*$ und $s \in S$.

$$\lambda\mu Prog(\Sigma)^{(w,s)} := \{e \in Exp(\Sigma)^{s_1 \to ... \to s_n \to s} \mid free(e) = \emptyset\}$$

ist die Menge der $\lambda\mu$-*Programme* über Σ vom Typ (w, s).

3. Für ein $e \in \lambda\mu Prog(\Sigma)^{(w,s)}$ ist die *Semantik*

$$\mathcal{M}_{\mathcal{A}}[\![e]\!] : \widehat{A^{s_1}} \times ... \times \widehat{A^{s_n}} \to \widehat{A^s}$$

definiert durch

$$\mathcal{M}_{\mathcal{A}}[\![e]\!](a_1, ..., a_n) = a :\Longleftrightarrow \mathcal{E}[\![e]\!]\rho_\perp(a_1)(a_2)...(a_n) = a$$

(mit $\rho_\perp(x) := \perp \; \forall x$)

Reduktionssemantik von $\lambda\mu$-Programmen

$\lambda\mu$-Berechnungsausdrücke: $\lambda\mu$-Ausdrücke mit Konstanten $a \in A$

Definition (Berechnungsausdruck)
Die Menge B der $\lambda\mu$-*Berechnungsausdrücke* über Σ und $\mathcal{A}$ ist induktiv als $Typ(S)$-sortierte Menge definiert durch

(0) $A^s \subseteq B^s$

1)-6) wie bei $Exp(\Sigma)$

Definition (Substitution)
Sei $y \in V^t$ und $u \in B^t$. Für $e \in B$ definieren wir die Substitution

$$e[y/u]$$

induktiv durch

(0) $a[y/u] := a$

(i) $f[y/u] := f$

(ii) $x[y/u] :=$ <u>if</u> $x = y$ <u>then</u> u <u>else</u> x

(iii) (<u>if</u> e_0 <u>then</u> e_1 <u>else</u> e_2 <u>fi</u>)$[y/u] :=$ <u>if</u> $e_0[y/u]$ <u>then</u> $e_1[y/u]$ <u>else</u> $e_2[y/u]$ <u>fi</u>

(v) $(e_1 e_2)[y/u] := (e_1[y/u] \; e_2[y/u])$

(iv) $(\lambda x.e)[y/u] :=$ <u>if</u> $x = y$ <u>then</u> $\lambda x.e$ <u>else</u> $\lambda x.(e[y/u])$

(vi) $(\mu x.e)[y/u] :=$ <u>if</u> $x = y$ <u>then</u> $\mu x.e$ <u>else</u> $\mu x.(e[y/u])$

Bemerkung: Nur freie Vorkommen von y werden ersetzt.

Beobachtung: Diese syntaktische Substitution ist semantisch korrekt, falls keine freie Variable von u nach Substitution gebunden wird.
Eine hinreichende Bedingung dafür ist: $bound(e) \cap free(u) = \emptyset$.

Substitutionslemma
Sei $y \in V^t, u \in B^t, e \in B$ und $\rho \in Env$. Wenn $bound(e) \cap free(u) = \emptyset$, so gilt

$$\mathcal{E}[\![e[y/u]]\!]\rho = \mathcal{E}[\![e]\!]\rho[y/\mathcal{E}[\![u]\!]\rho]$$

Bemerkung: Dieses Lemma schlägt die Brücke zwischen denotationeller und Operationssemantik.

Definition (Reduktionsregeln für $\lambda\mu$-Berechnungsausdrücke)

(i) *Konstantenreduktion*: [nicht in reinem λ-Kalkül]
$(...((f\ a_1)a_2)...a_n) \to f_{\mathcal{A}}(a_1, ..., a_n) \in A$

(ii) *Verzweigungsreduktion*:
$\underline{\text{if}}$ true $\underline{\text{then}}$ u_1 $\underline{\text{else}}$ $u_2 \to u_1$
$\underline{\text{if}}$ false $\underline{\text{then}}$ u_1 $\underline{\text{else}}$ $u_2 \to u_2$
(if nicht strikt)

(iii) *β-Reduktion* (*Kopierregel*):
$(\lambda x.e\ u) \to e[x/u]$,
falls $bound(e) \cap free(u) = \emptyset$

(iv) *Fixpunkt-Reduktion*:
$\mu x.e \to e[x/\mu x.e]$,
falls $(free(e) \setminus \{x\}) \cap bound(e) = \emptyset$ [„Abwickeln“ entspricht einmaligem Anwenden des Fixpunktoperators]

(v) *α-Konversion*: Umbenennung von gebundenen Variablen:
$\lambda x.e \to \lambda y.e[x/y]$, falls $y \notin free(e) \cup bound(e)$
$\mu x.e \to \mu y.e[x/y]$, falls $y \notin free(e) \cup bound(e)$

[KOMMENTAR: *Higher Order Konstrukte sind für uns nur Hilfsobjekte: Programme starten mit First Order Objekten und liefern auch ein solches als Ergebnis.*]

(** Folie 5.3 ** Beispiel: Fakultät 1!)

Definition (Reduktionsrelation)
Die *Reduktionsrelation* $\Rightarrow \subseteq B \times B$ ist induktiv definiert durch

(i) Wenn $e \to e'$, dann $e \Rightarrow e'$

(ii) Wenn $e \in B$, dann $e \Rightarrow e$

(iii) Wenn $e_i \Rightarrow e'_i$ $(1 \leq i \leq 3)$, dann

- $\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3\ \underline{\text{fi}} \ \Rightarrow\ \underline{\text{if}}\ e'_1\ \underline{\text{then}}\ e'_2\ \underline{\text{else}}\ e'_3\ \underline{\text{fi}}$
- $\lambda x.e_1 \Rightarrow \lambda x.e'_1$
- $(e_1\ e_2) \Rightarrow (e'_1\ e'_2)$
- $\mu x.e_1 \Rightarrow \mu x.e'_1$

Satz (Church-Rosser-Eigenschaft)
$\Rightarrow$ ist *konfluent*, d.h. für $u, u_1, u_2 \in B$ mit $u \Rightarrow^* u_1$ und $u \Rightarrow^* u_2$ gibt es ein $u' \in B$ mit $u_1 \Rightarrow^* u'$ und $u_2 \Rightarrow^* u'$.

Definition (Reduktionssemantik)
Für ein $\lambda\mu$-Programm $e \in \lambda\mu Prog(\Sigma)^{(s_1 \ldots s_n, s)}$ ist die *Reduktionssemantik* von e bezüglich $\mathcal{A}$,

$$\mathcal{M}^{red}_{\mathcal{A}}[\![e]\!] : A^{s_1} \times \ldots \times A^{s_n} \dashrightarrow A^s$$

definiert durch:

$$\mathcal{M}^{red}_{\mathcal{A}}[\![e]\!](a_1, \ldots, a_n) = a :\Longleftrightarrow (\ldots((e\ a_1)a_2)\ldots a_n) \Rightarrow^* a$$

Satz (Korrektheit und Vollständigkeit der Reduktionssemantik)
Für jedes $e \in \lambda\mu Prog(\Sigma)^{(w,s)}$ und $(a_1, \ldots, a_n, a) \in A^{ws}$ gilt:

$$\mathcal{M}_{\mathcal{A}}[\![e]\!](a_1, \ldots, a_n) = a \iff \mathcal{M}^{red}_{\mathcal{A}}[\![e]\!](a_1, \ldots, a_n) = a$$

Beweis: Substitutionslemma.

[KOMMENTAR: *Relative Berechenbarkeitstheorie / Programmschemata: über* beliebiger *Basis gilt:*

$$WHILE \subset GOTO \subset \{\ \textit{rek. Ber.}\ \} \subset \{\ \textit{Selbstanwendung}\ \}$$

(alle Inklusionen sind echt: $\neq$)
Herkömmliche Berechenbarkeitstheorie: über $\mathbb{N}$: alles „=“.]

[KOMMENTAR: *Problem bei β-Reduktion:* „variable capture“: *Term, der eingesetzt wird, enthält freie Variablen, die nach Einsetzen im entstandenen Term gebunden sind.*]

Die LO-Auswertungsstrategie

Vereinfachung: Beschränkung auf geschlossene $\lambda\mu$-Berechnungsausdrücke vom Basistyp
$\rightarrow$ Vermeidung von Variablenkonflikten; somit ist keine α-Konversion nötig.

Beispiel: Für $e = (\lambda x.u_1\ u_2)$ mit $free(e) = \emptyset$ gilt stets $bound(u_1) \cap free(u_2) = \emptyset$, weil $free(u_2) \subseteq free(e) = \emptyset$.
Für $e = \mu x.u$ mit $free(e) = \emptyset$ gilt stets $(free(u) \setminus \{x\}) \cap bound(u) = \emptyset$, weil

$free(u) \setminus \{x\} = \emptyset$.

Beachte: Reduktion führt nicht aus dem Bereich geschlossener $\lambda\mu$-Berechnungsausdrücke heraus.

Definition (LO-Strategie)
(*call by name*, *normal order*, *lazy evaluation*)
Sei $gB := \{e \in B^{basis} \mid free(e) = \emptyset\}$ die Menge der geschlossenen $\lambda\mu$-Ausdrücke vom Basistyp. Wir definieren $\Rightarrow_{LO} \subseteq gB \times gB$ durch

(i) Wenn $u_1 \to u_2$ (keine α-Konversion), dann $u_1 \Rightarrow_{LO} u_2$

(ii) $(...((...(f\ a_1)...a_{i-1})u_i)...u_n) \Rightarrow_{LO} (...((...(f\ a_1)...a_{i-1})u_i')...u_n)$, falls „linke Seite“$\in gB$, $(u_i \notin A)$, $u_i \Rightarrow_{LO} u_i'$.

(iii) if u then u_1 else u_2 fi $\Rightarrow_{LO}$ if u' then u_1 else u_2 fi, falls $u \Rightarrow_{LO} u'$.

(iv) $(u_1\ u_2) \Rightarrow_{LO} (u_1'\ u_2)$, falls $u_1 \Rightarrow_{LO} u_1'$.

Lemma: $\Rightarrow_{LO}$ ist deterministisch und korrekt bezüglich der Fixpunktsemantik.

[KOMMENTAR: $\mathcal{E}[\![x_i]\!]\rho := \rho(x) \Rightarrow$ *Nichtstriktheit; es gibt auch andere Möglichkeiten. Vergleiche: wie wurden Projektionen / Konstanten interpretiert?* $\Rightarrow$ *strikt / nichtstrikt.*]

5.2 Der ungetypte Lambda-Kalkül

A. *Church* (1936): Präzisierung des Berechenbarkeitsbegriffs
Natürliche Zahlen, boolesche Ausdrücke, Verzweigung etc. sind durch Lambda-Ausdrücke kodierbar.

Merkmale: ungetypte Variablen, Abstraktion, Applikation

Sei V eine abzählbare Menge von Variablen.

Definition (Lambda-Ausdruck)
Die Menge Λ der *λ-Ausdrücke* sei definiert durch

- $V \subseteq \Lambda$
- Wenn $x \in V, e \in \Lambda$, dann $\lambda x.e \in \Lambda$
- Wenn $e_1, e_2 \in \Lambda$, dann $(e_1\ e_2) \in \Lambda$

Ungetypte Applikation erlaubt *Selbstapplikation* $(e\ e)$ und dadurch eine Darstellung des Fixpunktoperators

$$Y := \lambda f.(\lambda x.(f(x\ x))\lambda x.(f(x\ x)))$$

(*kombinatorische Logik*; Curry: Kombinatoren S,K,I)

(** Folie 5.4 ** Beispiel: Fixpunktoperator durch Selbstapplikation)
[KOMMENTAR: *beliebte Prüfungsfrage*]

Reduktionsregeln: β-Reduktion, α-Konversion und „*Extensionalität*“: η-Konversion: $\lambda x.(e\ x) \rightarrow e$, falls $x \notin free(e)$

Literatur: Barendregt: The lambda calculus – its syntax and semantics, North Holland 1984

Bemerkung: Semantik?
Problem: Was bedeutet $(e\ e)$?
D_∞: *projektiver Limes* von $D_0, D_1 = [D_0 \rightarrow D_0], D_{i+1} = [D_i \rightarrow D_i]$
$D_\infty = \{(d_0, d_1, ...) \mid d_i \in D_i\}$
Es gilt: $D_\infty \cong [D_\infty \rightarrow D_\infty]$.
In diesem Raum kann $(e\ e)$ als $(\overline{e}\ e)$ mit $e \mapsto \overline{e}$ erklärt werden.

5.3 Kombinatorprogramme

$\lambda\mu$-Programme: beliebige Schachtelung von Abstraktion und Rekursion
Entschachtelung durch

- mehrfache Rekursion: $\mu(x_1, ..., x_n).(e_1, ..., e_n)$
- λ-Lifting: Abstraktionen herausziehen

Resultat: Normalform mit einer äußeren mehrfachen Rekursion, ohne innere Abstraktion.

Syntax von Kombinatorprogrammen:
Sei $\Sigma = \langle S, \Omega \rangle$ eine Signatur mit *bool* $\in S$. Für $X \subseteq V$ definieren wir die Menge $AExp(\Sigma, X)$ der *applikativen Ausdrücke* über Σ und X als kleinste $Typ(S)$-sortierte Menge E mit

(i) $X \cup \Omega \subseteq E$

(ii) if e_0 then e_1 else e_2 fi $\in E^t$, falls $e_0 \in E^{bool}$ und $e_1, e_2 \in E^t$

(iii) $(e_0\ e_1) \in E^{t_1}$, falls $e_0 \in E^{t_0 \rightarrow t_1}$ und $e_1 \in E^{t_0}$

Konvention: $(e_0\ e_1...e_n) := (...(e_0\ e_1)...e_n)$

Definition (Kombinatorprogramm)
Ein Gleichungssystem der Form (GS)
$F_1 x_{11} ... x_{1n_1} = e_1$
...

$F_r x_{t1} ... x_{rn_r} = e_r$
heißt *Kombinatorprogramm* über Σ, falls gilt: $F_i \in V^{t_{i1} \to t_{i2} \to ... \to t_{in_i} \to t_i}$, $x_{ij} \in V^{t_{ij}}$, $e_i \in AExp^{t_i}(\Sigma, \{F_1, ..., F_r, x_{i1}, ..., x_{in_i} \mid 1 \leq i \leq r\})$ und $t_{11}, ..., t_{1n_1}, t_1 \in S$ (Basistypen)
Bezeichnung: (GS) $\in KProg(\Sigma)$.

Beachte: Unterschied zu $Rek(\Sigma)$: hier „higher order".

Fixpunktsemantik: analog first order (nicht-strikt)

Reduktionssemantik:
$\mathcal{A} = \langle \widehat{A}; \alpha \rangle$ sei eine Σ-Interpretation.
Berechnungsterme: $B := AExp(\Sigma, \{F_1, ..., F_r\}, A)$
[KOMMENTAR: *Die Elemente von A treten an die Stelle der Variablen.*]
(keine Argument-Variablen, aber Konstanten aus A)

Reduktionsregeln:

- $(f\ a_1 ... a_n) \to f_{\mathcal{A}}(a_1, ..., a_n)$ (δ-Redex)
- $(F_i\ u_1 ... u_{n_i}) \to e_i[x_{i1}/u_1, ..., x_{in_i}/u_{n_i}]$
- if true then ... $\to$ ...

Definition (LO-Auswertungsstrategie)

- $\to \subseteq \Rightarrow_{LO}$
- $(f\ a_1 ... a_{i-1} u_i ... u_n) \Rightarrow_{LO} (f\ a_1 ... a_{i-1} u_i' u_{i+1} ... u_n)$, falls $u_i \Rightarrow_{LO} u_i'$
- if u ... $\Rightarrow_{LO}$ if u' ..., falls $u \Rightarrow_{LO} u'$
  [KOMMENTAR: *Beachte: keine spezielle Regel* $(F...) \Rightarrow_{LO} (F...)$, *da F reduziert wird.*]
- $(u_0\ u_1) \Rightarrow_{LO} (u_0'\ u_1)$, falls $u_0 \Rightarrow_{LO} u_0'$

[KOMMENTAR: *Diese Auswertungsstrategie ist wieder deterministisch und vollständig.*]

Reduktionssemantik:
... $F_i(a_1 ... a_{n_i}) \Rightarrow_{LO}^* a$

Bemerkung: $Rek(\Sigma) \rightsquigarrow KProg(\Sigma) \sim \lambda\mu Prog$
(Die rekursiven Funktionsdefinitionen erster Ordnung sind in Kombinatorprogramme übersetzbar, und diese sind äquivalent zur Menge der $\lambda\mu$-Programme.
[KOMMENTAR: *Vergleiche Satz von* Kleene *für endliche Automaten / reguläre Ausdrücke:* $\mu \approx ()^*$.]
(** Folie 5.5 ** Beispiel: Entschachtelung von $\lambda\mu$-Programmen)
(** Folie 5.6 ** Beispiel: Kombinatorreduktion (fib))

Die *Kombinator-Graphmaschine*:
Implementierung der LO-Strategie
Probleme:

1. Verzögerte Argumentberechnung. Bisher:
   - Stacktechnik mit X-Blöcken bei Rekursion erster Ordnung
   - Heap-Technik mit Suspensionsknoten bei Rekursion mit Konstruktoren
2. Funktionen höherer Ordnung, partielle Applikationen $\rightarrow$ Stack-Technik mit F-Blöcken unzureichend

Lösung: F-Blöcke ebenso wie verzögerte Argumentberechnungen in den Heap verlagern; statt Funktionskeller nun *Kontroll-Keller*.

Graphknoten:

1. *Applikationsknoten* (*Funktionsknoten*) $(AP, \varphi, arg_1, ..., arg_p, i)$
   mit $\varphi \in \{F_1, ..., F_r\}, p \leq rg(\varphi), i = rg(\varphi) - p$
   oder $\varphi \in \Omega^{(s_1 \dots s_n, s)}, 0 < n, p \leq n, i = n - p$
   und $arg_r \in A \cup GAdr$. $\quad (rg(F_i) = n_i)$
2. *Suspensionsknoten* (*Argumentknoten*) $(SUSP, pa, arg_1, ..., arg_n, n)$
   mit $pa \in PAdr, arg_r \in A \cup GAdr, n \in I\!N$

Zustandsraum: $Z := BZ \times DK \times GR \times KK$ mit

- $BZ := PAdr = I\!N$ Programmadressen
- $DK := (A \cup GAdr)^*$ Datenkeller (Spitze rechts)
- GR mit Applikations- und Suspensionsknoten
- $KK := (GAdr \times PAdr)^*$ Kontroll-Keller (Spitze links)
  [KOMMENTAR: *GAdr: Umgebung, PAdr: Rücksprungadresse*]

(** Folie 5.7 ** Graphmaschine)
(** Folien 5.8, 5.9 ** Graphmaschine Befehlssatz)

Anfangs- und Endzustand:
Eingabe: $(a_1, ..., a_{n_1}) \in A^w$ bestimmt den Anfangszustand $(1, \varepsilon, g_0, k_0)$ mit

- $g_0(0) = (AP, F_1, a_1, ..., a_{n_1}, 0),\ free(g_0) = 1$
- $k_0 = (0, 0)$

Endzustand: $(0, a, g, \varepsilon) \mapsto a$ Ergebnis

(** Folie 5.10 ** Codeerzeugung für Kombinatorprogramme)
(** Folie 5.11 ** Beispiel)
(** Folie 5.12 ** Beispielrechnung)

6 Typkonzepte

Inhalt:

- *Typdeklarationen* in *Gofer*
- *Polymorphie*
- *Monade*

6.1 Typdeklarationen

Vordefinierte Typen: Int, Float, Char, Bool
Vordefinierte Typkonstruktoren:

- $T_1 \rightarrow T_2$ (Funktionen von T_1 nach T_2)
- (T_1, T_2) (Paare von Elementen aus T_1 und T_2)
- $[T]$ (Listen über dem Elementtyp T)

Typabkürzung (ohne Rekursion)

```
type String = [Char]
type Position = (Float,Float)
type Pair a b = (a,b)
```

($\rightarrow$ parametrische Polymorphie; a,b *Typvariablen*)

Algebraische Datentypen (mit Rekursion):

- Aufzählungstypen
  ```
  data Color = Red | Yellow | Green
  ```
  (Color ist Typkonstruktor, Red, Yellow, Green sind Datenkonstruktoren.)
  ```
  data Bool = True | False
  ```
- Verbundtypen
  ```
  data Position = Position (Float,Float)
  ```
  (vorne: Typkonstruktor, hinten: Datenkonstruktor)
- Variante Verbunde und rekursive Datentypen
  ```
  data List a = NIL | Cons a (List a)
  ```
  (Typkonstruktor List (1-stellig), Datenkonstruktoren NIL, Cons)
  ```
  data BTree a = EmptyTree | Branch a (BTree a) (BTree a)
  leaves :: BTree -> Int
  leaves EmptyTree = 0
  leaves (Branch _ EmptyTree EmptyTree) = 1
  leaves (Branch _ l r) = leaves l + leaves r
  ```

6.2 Polymorphie

Definition (polymorph)
Eine Funktion heißt *polymorph*, wenn ihre Parameter verschiedene Typen annehmen dürfen.
Dabei gibt es zwei Arten von Polymorphie:

a) *parametrische Polymorphie*, z.B. Listenoperationen

b) *Ad-hoc-Polymorphie*: Überladen von Operatoren

Genauer:

a) Typvariablen in einem Typ $\rightarrow$ polymorpher Typ
Jeder Typ kann für eine Typvariable eingesetzt werden. Häufig bei algebraischen Datentypen.
Beispiel:

```
length :: [a] -> Int
(++) :: [a] -> [a] -> [a]
head :: [a] -> a
tail :: [a] -> [a]
fst :: (a,b) -> a
scd :: (a,b) -> b
id :: a -> a
if_then_else :: Bool -> a -> a -> a
```

b) Die Gleichheit ist nicht parametrisch-polymorph!
```
(==) :: a -> a -> Bool
```
(falsch! weil die Gleichheit von Funktionen nicht entscheidbar ist: wegen des Halteproblems von Turingmaschinen)
Lösung: Einschränkung auf Typklassen:

Die Typklasse Eq: Typen mit (==), (/ =)

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x/=y = not(x==y)
  (==), (/=) :: Eq a => a -> a -> Bool
```

(`Eq a =>` ist der Kontext.)
Exemplare der Typklasse Eq:

```
instance Eq Int where (==) = primEqInt
instance Eq a => Eq [a] where
                    []==[] = True
                    (x:xs)==(y:ys) = x==y && xs==xs
                    _==_ = False
instance (Eq a,Eq b) => Eq (a,b) where
                    (x,y)==(x',y') = x==x' && y==y'
```

Die Unterklasse Ord von Eq
Vergleichsoperationen:

```
class Eq a => Ord a value
      (<),(>),(<=),(>=) :: a -> a -> Bool
      min,max :: a -> a -> a
      x<y  = x<=y && x/=y
      x>=y = y<=x
      x>y  = y<=x && y/=x
      max x y | x<=y = y
                otherwise = x
      min x y | x<=y = x
                otherwise = y

instance OrdInt where (<=)=primLeInt
instance (Ord a,Ord b) => Ord (a,b) where
        (x1,x2)<=(y1,y2) = x1<y1 || x1==y1 && x2<=y2
instance Ord a => Ord [a] where
        []<=_ = True
        (_:_)<=[] = False
        (x:xs)<=(y:ys) = x<y || x==y && xs<=ys
```

Beispiel: `sort :: Ord a => [a] -> [a]`

6.3 Monaden

- parametrisierter Datentyp zur Kapselung von Berechnungen
- uniforme Behandlung von zusätzlichen Berechnungen durch geeignete Abstraktion

```
type M a
  result :: a -> M a
  bind :: M a -> (a -> M b) -> M b
```

mit Typvariable a und variablem Typkonstruktor M
Eine Instanz von M muß folgenden Gleichungen genügen:

1. `result x ‘bind‘ f = f x`
2. `m ‘bind‘ result = m`
3. `(m ‘bind‘ f) ‘bind‘ g = m ‘bind‘ (λx -> f x ‘bind‘ g)`
   (Assoziativgesetz)

Programmmodifikation durch Ändern der Monade
→ Verbesserung von Modularität, Lesbarkeit, Wartbarkeit

Beispiel: Auswertung artihmetischer Ausdrücke

```
data AExp = Var Variable
          | Const Value
          | Plus AExp AExp
          | Let Variable AExp AExp
type Value = Int
type Variable = String
type Env = [(Variable,Value)]

eval :: AExp -> Env -> M Value
eval (Var v) e = lookup v e
eval (Const c) e = result c
eval (Plus a1 a2) e = eval a1 e `bind` (\l v1 -> eval a2 e `bind`
                                       (\l v2 -> add v1 v2))
eval (Let v a1 a2) e = eval a1 e `bind` (\l x -> eval a2
                                           (update e v x))
```

Hilfsfunktionen:

```
lookup :: Variable -> Env -> M Value
lookup v' ((v,x):e) | v'==v     = result x
                    | otherwise = lookup v' e
lookup v' _                     = raise v'
```

(Fehlerbehandlung)

```
addV :: Value -> Value -> M Value
addV x y = result (x+y)

update :: Env -> Variable -> Value -> Env
update e v x = (v,x):e
```

Zum Testen von eval:

```
display :: Text a => M a -> String
test exp = display (eval exp[])
```

1. Monade: Identität
Nur ursprüngliche Werte, keine Multiplikation

```
type I a = a
result = id
bind x f = f x
raise i = error "lookup failed"
add = addV
display = show

? test (Plus (Const 1) (Const 2))
3
? test (Var "x")
Program error: lookup failed
```

2. Monade: Fehlerbehandlung
Fehlerkontrolle durch das Programm selbst:

```
data E a = Error String | Ok a
result x = Ok x
bind m f = case m of
           Error s -> Error s
           Ok x    -> f x
raise s = Error s
add = addV
display (Error s) = "Error: "++s
display (Ok x)    = "Ok: "++show x
```

(Voraussetzung: a vom Typ text; Ausgabe möglich)
Mit `M a = E int` ergibt sich:

```
? test (Plus (Const 1) (Const 2))
Ok: 3
? test (Var "x")
Error: x
```

3. Monade: Zustandstransformation
Ziel: Zählen der ausgeführten Additionen durch „Mitschleppen" einer Zustandskomponente

```
data ST a = Int -> (a,Int)         % ST = states
result x = \l z.(x,z)
bind m f = \l z.let (x',z') = m z
                in f x' z'
add :: Value -> Value -> ST Value
add x y = incr `bind` (\l k.addV x y)
```

mit

```
incr :: ST ()                      % (): trivialer Typ
incr = \l z. ( (), z+1)
raise x = error "lookup failed"
display m = case (m 0) of
            (x,z) -> "Count: "++show z    % z = # durchgef. Add.
                     ++ "Value: "++show x
```

Mit `M a = ST Int` ergibt sich:

```
? test (Plus (Const 1) (Const 2))
Count: 1  Value: 3
? test (Let "x" (Plus (Const 8) (Const 13))
                (Plus (Var "x") (Var "x")))
Count: 2  Value: 42
```

Index