Funktionale Programmierung und Typtheorie

Francesco Kriegel

TU Dresden
Fakultät Mathematik
Institut Algebra

WS 2008 / 2009

16. April 2010

Inhaltsverzeichnis

1 Der λ-Kalkül

1.1 Ungetypter Lambda-Kalkül

1.1.1 Definition und Syntax

Definition 1.1 (λ-Term)

Für eine mindestens abzählbar unendliche Menge $\mathbf{V}$ von *Variablen* ist die Menge $\mathbf{\Lambda}$ der *λ-Terme* über $\mathbf{V}$ induktiv definiert durch:

(i) $x \in \mathbf{V} \implies x \in \mathbf{\Lambda}$

(ii) $A \in \mathbf{\Lambda} \implies (\lambda x.A) \in \mathbf{\Lambda}$ (*Abstraktion*)

(iii) $A, B \in \mathbf{\Lambda} \implies (AB) \in \mathbf{\Lambda}$ (*Applikation*)

Bemerkung 1.2 Für die Darstellung von λ-Termen benutzen wir folgende Notationen:

(a) Die äußersten Klammern können weggelassen werden.

(b) Wir schreiben $A \equiv B$, falls die λ-Terme A und B *syntaktisch äquivalent* sind.

(c) Der Rumpf einer Abstraktion geht so weit wie möglich nach rechts, d.h.

$$\lambda x.(AB) \equiv \lambda x.AB$$

für alle $x \in \mathbf{V}$ und $A, B \in \mathbf{\Lambda}$.

(d) Abstraktion ist rechtsassoziativ und die λ-Bindungen werden zusammengefasst, d.h.

$$\lambda x_1(\lambda x_2(\dots(\lambda x_n.A)\dots)) \equiv \lambda x_1.\lambda x_2.\dots.\lambda x_n.A \equiv \lambda x_1 x_2 \dots x_n.A$$

für alle $x_1, x_2, \dots, x_n \in \mathbf{V}$ und $A \in \mathbf{\Lambda}$.

(e) Applikation ist linksassoziativ, d.h.

$$(AB)C \equiv ABC$$

für alle $A, B, C \in \mathbf{\Lambda}$.

Definition 1.3 (freie und gebundene Variable)

(a) Für einen λ-Term A ist $\mathbf{FV}(A)$ die Menge der *freien Variablen*, und $\mathbf{FV}(A)$ ist induktiv definiert durch

(i) $\mathbf{FV}(x) = \{x\}$

(ii) $\mathbf{FV}(\lambda x.A) = \mathbf{FV}(A) \setminus \{x\}$

(iii) $\mathbf{FV}(AB) = \mathbf{FV}(A) \cup \mathbf{FV}(B)$

für alle $x \in \mathbf{V}$ und $A, B \in \mathbf{\Lambda}$.

(b) Eine Variable x heißt *frei* in A, wenn $x \in \mathbf{FV}(A)$.

(c) Ein λ-Term A mit $\mathbf{FV}(A) = \emptyset$ heißt *geschlossener λ-Term* oder *Kombinator*.

(d) $\Lambda^0 := \{A \in \Lambda \mid \mathbf{FV}(A) = \emptyset\}$ ist die Menge aller Kombinatoren.

(e) Entsprechend ist $\mathbf{BV}(A)$ die Menge der *gebundenen Variablen*, und $\mathbf{BV}(A)$ ist induktiv definiert durch

(i) $\mathbf{BV}(x) = \emptyset$

(ii) $\mathbf{BV}(\lambda x.A) = \mathbf{BV}(A) \cup \{x\}$

(iii) $\mathbf{BV}(AB) = \mathbf{BV}(A) \cup \mathbf{BV}(B)$

Definition 1.4 (Teilterme)

Für einen λ-Term A ist $\mathbf{Sub}(A)$ die Menge der *Teilterme*, und diese ist induktiv definiert durch:

(i) $\mathbf{Sub}(x) = \{x\}$

(ii) $\mathbf{Sub}(\lambda x.B) = \{\lambda x.B\} \cup \mathbf{Sub}(B)$

(iii) $\mathbf{Sub}(BC) = \{BC\} \cup \bigcup_{D_1 \ldots D_n = BC} \bigcup_{i=1}^{n} \mathbf{Sub}(D_i)$

Ein λ-Term B heißt *Teilterm* von A, falls $B \in \mathbf{Sub}(A)$, und wir schreiben dafür auch $B \subset A$.

Definition 1.5 (Substitution)

Für eine Variable x und λ-Terme A, B ist $A[x := B]$ die *Substitution* von x durch B in A, und ist induktiv definiert durch:

(i) $x[x := B] \equiv B$

(ii) $y[x := B] \equiv y$

(iii) $(\lambda x.C)[x := B] \equiv \lambda x.C$

(iv) $(\lambda y.C)[x := B] \equiv \lambda y.(C[x := B])$ für $x \notin \mathbf{Sub}(C)$ oder $y \notin \mathbf{FV}(B)$

(v) $(\lambda y.C)[x := B] \equiv \lambda z.(C[y := z][x := B])$ sonst

(vi) $(CD)[x := B] \equiv (C[x := B])(D[x := B])$

Dabei sind die Variablen x, y, z paarweise verschieden und $z \notin \mathbf{Sub}(B) \cup \mathbf{Sub}(C)$.

Satz 1.6 (Substitutionslemma) Es gilt

$$A[x := B][y := C] \equiv A[y := C][x := B[y := C]]$$

für Variablen x, y und λ-Terme A, B, C mit $x \neq y$ und $x \notin \mathbf{FV}(C)$.

Definition 1.7 (λ-Kalkül)

Die Theorie $\boldsymbol{\lambda}$ hat als Formeln Gleichungen

$$A = B$$

für λ-Terme $A, B \in \mathbf{\Lambda}$. Wir nennen $\boldsymbol{\lambda}$ auch *λ-Kalkül*. Es gelten folgende Axiome für alle $x, y \in \mathbf{V}$ und $A, B, C \in \mathbf{\Lambda}$:

(i) $A = A$ (reflexiv)

(ii) $A = B \implies B = A$ (symmetrisch)

(iii) $A = B \wedge B = C \implies A = C$ (transitiv)

(iv) $A = B \implies AC = BC \wedge CA = CB$

(v) $A = B \implies \lambda x.A = \lambda x.B$ (ξ)

(vi) $\lambda x.A = \lambda y.A[x := y]$ für $y \notin \mathbf{FV}(A)$ (α)

(vii) $(\lambda x.A)B = A[x := B]$ für $\mathbf{BV}(A) \cap \mathbf{FV}(B) = \emptyset$ (β)

(viii) $\lambda x.Ax = A$ für $x \notin \mathbf{FV}(A)$ (η)

Die Beweisbarkeit einer Gleichung im λ-Kalkül symbolisieren wir mit $\boldsymbol{\lambda} \vdash A = B$ oder einfach $A = B$, und dann heißen A und B *konvertierbar*. Es gilt stets $A \equiv B \implies A = B$, aber nicht umgekehrt.

1.1.2 Beispiele

BOOLEsche Terme

$$\mathsf{true} \equiv \lambda ab.a$$

$$\mathsf{false} \equiv \lambda ab.b$$

$$\mathsf{or} \equiv \lambda ab.a\,\mathsf{true}\,b$$

$$\mathsf{and} \equiv \lambda ab.ab\,\mathsf{false}$$

$$\mathsf{not} \equiv \lambda a.a\,\mathsf{false}\,\mathsf{true}$$

$$\mathsf{ifthenelse} = \mathsf{ite} \equiv \lambda abc.abc$$

Satz 1.8 Es gelten $\mathsf{ifthenelse}\,\mathsf{true}\,AB \equiv A$ und $\mathsf{ifthenelse}\,\mathsf{false}\,AB \equiv B$.

Paare

$$\mathsf{pair} \equiv \lambda pqf.fpq$$

$$\mathsf{fst} \equiv \lambda p.p\,\mathsf{true}$$

$$\mathsf{snd} \equiv \lambda p.p\,\mathsf{false}$$

Listen

$$\mathsf{cons} \equiv \mathsf{pair}$$

$$\mathsf{hd} \equiv \mathsf{fst}$$

$$\mathsf{tl} \equiv \mathsf{snd}$$

$$\mathsf{nil} \equiv \lambda f.\mathsf{true}$$

$$\mathsf{null} \equiv \lambda l.l(\lambda ht.\mathsf{false})$$

Zahlen

Für $F, X \in \mathbf{\Lambda}$ und $n \in \mathbb{N}$ definieren wir die *Potenz* $F^n X$ induktiv durch:

(i) $F^0 X = X$

(ii) $F^{n+1} X = F(F^n X)$

Definition 1.9 (CHURCH-Zahlen)

Die CHURCH-*Zahlen* C_n sind definiert als

$$\mathsf{C}_n \equiv \lambda f x.f^n x$$

mit den Operationen:

(i) Nulltest $\mathsf{iszero} \equiv \lambda n.n(\lambda x.\mathsf{false})\mathsf{true}$

(ii) Nachfolgerbildung $\mathsf{succ} \equiv \lambda n f x.f(nfx)$

(iii) Addition $\mathsf{add} \equiv \lambda m n f x.mf(nfx)$

(iv) Multiplikation $\mathsf{mult} \equiv \lambda m n f.m(nf)$

(v) Exponentiation $\mathsf{exp} \equiv \lambda m n.mn$

Satz 1.10 Es gelten:

(i) $\mathsf{iszero}\mathsf{C}_0 \equiv \mathsf{true}$ und $\mathsf{iszero}(\mathsf{succ}\mathsf{C}_n) \equiv \mathsf{false}$

(ii) $\mathsf{succ}\mathsf{C}_n \equiv \mathsf{C}_{n+1}$

(iii) $\mathsf{add}\mathsf{C}_m\mathsf{C}_n \equiv \mathsf{C}_{m+n}$ und $\mathsf{add}\mathsf{C}_m\mathsf{C}_n \equiv \mathsf{succ}^m\mathsf{C}_n$

(iv) $\mathsf{mult}\mathsf{C}_m\mathsf{C}_n \equiv \mathsf{C}_{m\cdot n}$ und $\mathsf{mult}\mathsf{C}_m\mathsf{C}_n \equiv (\mathsf{add}\mathsf{C}_n)^m\mathsf{C}_0$

(v) $\mathsf{exp}\mathsf{C}_m\mathsf{C}_n \equiv \mathsf{C}_{n^m}$ und $\mathsf{exp}\mathsf{C}_m\mathsf{C}_n \equiv (\mathsf{mult}\mathsf{C}_n)^m\mathsf{C}_1$

$$\mathsf{pred} = \lambda n.\mathsf{snd}(nh(\mathsf{pair}\mathsf{C}_0\mathsf{C}_0))$$
$$h = \lambda p.\mathsf{pair}(\mathsf{succ}(\mathsf{fst}p))(\mathsf{fst}p)$$

1.1.3 Reduktion und Normalform

Definition 1.11 (Λ-Hülle, ρ-Reduktion, ρ-Äquivalenz)

Sei ρ eine binäre Relation auf Λ.

(a) Die Λ-*Hülle* $\to_\rho$ von ρ ist eine binäre Relation auf Λ und wird induktiv definiert durch:

(i) $(A, B) \in \rho \implies A \to_\rho B$

(ii) $A \to_\rho B \implies AC \to_\rho BC \wedge CA \to_\rho CB$

(iii) $A \to_\rho B \implies \lambda x.A \to_\rho \lambda x.B$

Für $A \to_\rho B$ sagen wir auch, A ist *in einem Schritt ρ-reduzibel* zu B, und A heißt *ρ-Redex* und B heißt *ρ-Kontraktum*.

(b) Die *ρ-Reduktion* $\Rightarrow_\rho$ ist die reflexive, transitive Hülle von $\to_\rho$, d.h. es gelten

(i) $A \to_\rho B \implies A \Rightarrow_\rho B$

(ii) $A \Rightarrow_\rho A$

(iii) $A \Rightarrow_\rho B \wedge B \Rightarrow_\rho C \implies A \Rightarrow_\rho C$

Für $A \Rightarrow_\rho B$ sagen wir auch, A ist *ρ-reduzibel* zu B.

(c) Die *ρ-Konvertibilität* ist die von $\to_\rho$ erzeugte Äquivalenzrelation, d.h. die symmetrische, transitive Hülle von $\Rightarrow_\rho$ und es gelten:

(i) $A \Rightarrow_\rho B \implies A =_\rho B$

(ii) $A =_\rho B \implies B =_\rho A$

(iii) $A =_\rho B \wedge B =_\rho C \implies A =_\rho C$

Für $A =_\rho B$ sagen wir auch, A und B sind *ρ-konvertibel*.

Wir verwenden obige Definitionen für diese drei Relationen:

$$\alpha := \{(\lambda x.A, \lambda y.A[x := y]) \mid x, y \in \mathbf{V} \wedge A \in \mathbf{\Lambda} \wedge y \notin \mathbf{FV}(A)\}$$

$$\beta := \{((\lambda x.A)B, A[x := B]) \mid x \in \mathbf{V} \wedge A, B \in \mathbf{\Lambda} \wedge \mathbf{BV}(A) \cap \mathbf{FV}(B) = \emptyset\}$$

$$\eta := \{(\lambda x.Ax, A) \mid x \in \mathbf{V} \wedge A \in \mathbf{\Lambda} \wedge x \notin \mathbf{FV}(A)\}$$

Definition 1.12 (β-Normalform)

(i) Ein λ-Term A ist in *β-Normalform*, wenn er keine β-Redexe als Teilterme enthält.

(ii) Ein λ-Term A hat eine *β-Normalform*, wenn es einen λ-Term B in β-Normalform mit $A =_\beta B$ gibt.

(iii) Ein λ-Term A heißt *schwach β-normalisierend*, falls eine endliche

Reduktionsfolge $A \to_\beta A_1 \to_\beta A_2 \to_\beta \ldots \to_\beta A_n$ existiert, sodass A_n eine β-Normalform ist.

(iv) Ein λ-Term A heißt *stark β-normalisierend*, falls jede Reduktionsfolge $A \to_\beta A_1 \to_\beta A_2 \to_\beta \ldots$ endlich ist.

Satz 1.13 (i) Nicht jeder λ-Term hat eine β-Normalform, d.h. nicht alle λ-Terme sind schwach normalisierend.

(ii) Es ist nicht entscheidbar, ob ein λ-Term eine β-Normalform besitzt.

Beweis: (i) Der λ-Term $(\lambda x.xx)(\lambda x.xx)$ hat keine β-Normalform.

(ii) Beweis über Halteproblem. ■

Lemma 1.14 Falls $A \in \mathbf{\Lambda}$ in β-Normalform ist, dann gilt

$$A \Rightarrow_\beta B \implies A = B.$$

Beweis: Für eine β-Normalform A gibt es kein B mit $A \to_\beta B$. Also kann $A \Rightarrow_\beta B$ nur wegen der Reflexivität von $\Rightarrow_\beta$ gelten, d.h. $A = B$. ■

Theorem 1.15 (Erstes CHURCH-ROSSER-Theorem) Seien A, B, C drei λ-Terme mit $A \Rightarrow_\beta B$ und $A \Rightarrow_\beta C$. Dann gibt es einen λ-Term D mit $B \Rightarrow_\beta D$ und $C \Rightarrow_\beta D$.

Folgerung 1.16 (i) Falls $A =_\beta B$, so existiert ein λ-Term C mit $A \Rightarrow_\beta C$ und $B \Rightarrow_\beta C$.

(ii) Falls A eine β-Normalform B hat, dann gilt $A \Rightarrow_\beta B$.

(iii) Ein λ-Term hat höchstens eine β-Normalform.

Definition 1.17 (LO- und LI-Reduktion)

(i) Eine *LO-Reduktion* ist eine Reduktion, bei der der am weitesten links stehende (*leftmost*) Redex, der in keinem anderen enthalten ist (*outermost*), reduziert wird. Andere Bezeichnungen sind *Normal-Order-Reduction* und *Call-by-Name-Mechanismus*.

(ii) Eine *LI-Reduktion* ist eine Reduktion, bei der der am weitesten links stehende (*leftmost*) Redex, der keinen anderen enthält (*in-*

nermost), reduziert wird. Andere Bezeichnungen sind *Applicative-Order-Reduction* und *Call-by-Value-Mechanismus*.

Theorem 1.18 (Zweites CHURCH-ROSSER-Theorem, Normalisierungstheorem) Wenn ein λ-Term A eine β-Normalform B hat, dann existiert eine Folge von LO-Reduktionen von A zu B.

1.1.4 Fixpunkte

Theorem 1.19 (i) Jeder λ-Term F besitzt einen *Fixpunkt* $X \in \mathbf{\Lambda}$, d.h. es gilt $FX = X$.

(ii) Es existiert ein *Fixpunktkombinator* $\mathsf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, sodass $F(\mathsf{Y}F) = \mathsf{Y}F$ für alle λ-Terme F gilt.

Satz 1.20 Die folgenden Terme sind ebenfalls Fixpunktkombinatoren:

(i) $\mathsf{Y}_n = \lambda f.X_n^n = \lambda f.\underbrace{X_n \ldots X_n}_{n\text{-mal}}$ mit $X_n = \lambda x_1 \ldots x_{n-1}.f(x_1 x_1 \ldots x_{n-1})$
für $n \geq 2$. Speziell ist $\mathsf{Y} = \mathsf{Y}_2$.

(ii) $\mathsf{Y}^* \equiv \lambda f.(\lambda gx.f(gg)x)(\lambda gx.f(gg)x)$

(iii) $\mathsf{Y}_n^* = \lambda f.B_n^n = \lambda f.\underbrace{B_n \ldots B_n}_{n\text{-mal}}$ mit $B_n = \lambda x_1 \ldots x_n.f(x_1 x_1 \ldots x_{n-1})x_n$
für $n \geq 2$. Speziell ist $\mathsf{Y}^* = \mathsf{Y}_2^*$.

(iv) $\Theta = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$

(v) $\$ \equiv £^{26} \equiv £££££££££££££££££££££££££$
mit $£ \equiv \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$

(vi) $\Theta_n = A_n^n = \underbrace{A_n \ldots A_n}_{n\text{-mal}}$ mit $A_n = \lambda x_1 \ldots x_n.x_n(x_1 x_1 \ldots x_n)$ für $n \geq 2$.
Speziell ist $\Theta = \Theta_2$ und $\$ = \Theta_{26}$.

Mit dem Fixpunkttheorem sind rekursive Definitionen möglich. Möchte man eine Funktion f durch den λ-Term A rekursiv definieren, d.h. $f = A$ und $f \in \mathbf{FV}(A)$, so erhält man die syntaktisch korrekte Definition vermöge

$$f = \mathsf{Y}(\lambda f.A).$$

Beispiel 1.21 (i) Eine intuitive Definition für die Fakultätsfunktion wäre beispielsweise

$$\mathsf{fac} = \lambda n.\mathsf{ite}(\mathsf{iszero}n)\mathsf{C}_1(\mathsf{mult}n(\mathsf{fac}(\mathsf{pred}n))).$$

Die Anwendung des Fixpunktkombinators ergibt

$$\mathsf{fac} = \mathsf{Y}(\lambda\mathsf{fac}.\lambda n.\mathsf{ite}(\mathsf{iszero}n)\mathsf{C}_1(\mathsf{mult}n(\mathsf{fac}(\mathsf{pred}n))))$$

(ii) Das Konkatenieren zweier Listen kann rekursiv definiert werden durch

$$\mathsf{append} = \lambda xy.\mathsf{ite}(\mathsf{null}x)y(\mathsf{cons}(\mathsf{hd}x)(\mathsf{append}(\mathsf{tl}x)y))$$

und syntaktisch korrekt muss die Definition also

$$\mathsf{append} = \mathsf{Y}(\lambda\mathsf{append}.\lambda xy.\mathsf{ite}(\mathsf{null}x)y(\mathsf{cons}(\mathsf{hd}x)(\mathsf{append}(\mathsf{tl}x)y)))$$

lauten.

(iii) Das Erzeugen einer unendlichen Liste, die nur aus Nullen besteht, kann durch

$$\mathsf{zeroes} = \mathsf{cons}\mathsf{C}_0\mathsf{zeroes}$$

rekursiv definiert werden. Mit dem Fixpunktkombinator haben wir

$$\mathsf{zeroes} = \mathsf{Y}(\lambda\mathsf{zeroes}.\mathsf{cons}\mathsf{C}_0\mathsf{zeroes}).$$

□

1.1.5 Curryfizierung

Aus der Mathematik ist bekannt, dass für Mengen A, B und C stets die Isomorphie

$$(A \times B) \to C \cong A \to (B \to C)$$

gilt, das bedeutet für eine Funktion $f\colon (A \times B) \to C$, dass es eine Funktion $f'\colon A \to (B \to C)$ mit $f'(a) = (b \mapsto f(a,b))$ gibt, sodass $f(a,b) = f'(a)(b)$ gilt. Das funktioniert auch im λ-Kalkül mit den Termen

$$\mathsf{curry}_2 = \lambda fxy.f(x,y)$$

und

$$\mathsf{uncurry}_2 = \lambda fp.f(\mathsf{fst}p)(\mathsf{snd}p)$$

und allgemein für n Variablen

$$\mathsf{curry}_n = \lambda f x_1 \dots x_n . f(x_1, \dots, x_n)$$

und

$$\mathsf{uncurry}_n = \lambda f p . f(\mathsf{fst} p) \dots (\mathsf{lst} p).$$

Damit lässt sich die λ-Abstraktion generalisieren zu

$$\lambda(x_1, \dots, x_n).A = \mathsf{uncurry}_n(\lambda x_1 \dots x_n . A)$$

und entsprechend ergibt sich die β-Reduktion dann zu

$$(\lambda(x_1, \dots, x_n).A)(B_1, \dots, B_n) = A[x_1 := B_1, \dots, x_n := B_n],$$

wobei keine der Variablen $x_1, \dots, x_n$ frei in $B_1, \dots, B_n$ vorkommen dürfen.

1.1.6 primitiv-rekursive und μ-rekursive Funktionen

Definition 1.22 (numerisch, λ-definierbar)

Eine Funktion $f\colon \mathbb{N}^n \to \mathbb{N}$ heißt *numerische Funktion*. Eine numerische Funktion f heißt *λ-definierbar*, falls es einen λ-Term F gibt, sodass für alle $x_1, \dots, x_n \in \mathbb{N}$

$$\mathsf{C}_{f(x_1,\dots,x_n)} = F\mathsf{C}_{x_1} \dots \mathsf{C}_{x_n}$$

gilt.

Definition 1.23 (Grundfunktion, Komposition, primitive Rekursion, primitiv-rek

(i) Die *Grundfunktionen* sind die drei numerischen Funktionen

- *Projektion* $(x_1, \dots, x_n) \mapsto x_i$
- *Nullfunktion* $(x_1, \dots, x_n) \mapsto 0$
- *Nachfolgerfunktion* $x \mapsto x + 1$.

(ii) Die *Komposition* $f = g[h_1, \dots, h_n]\colon \mathbb{N}^m \to \mathbb{N}$ von numerischen Funktionen $g\colon \mathbb{N}^n \to \mathbb{N}$ und $h_1, \dots, h_n\colon \mathbb{N}^m \to \mathbb{N}$ ist definiert durch

$$f(x_1, \dots, x_m) := g(h_1(x_1, \dots, x_m), \dots, h_n(x_1, \dots, x_m)).$$

(iii) Die *primitive Rekursion* $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ zweier numerischer Funk-

tionen $g\colon \mathbb{N}^{n+2} \to \mathbb{N}$ und $h\colon \mathbb{N}^n \to \mathbb{N}$ ist definiert durch

$$f(0, y_1, \ldots y_n) := h(y_1, \ldots, y_n)$$

$$f(x+1, y_1, \ldots y_n) := g(f(x, y_1, \ldots y_n), x, y_1, \ldots y_n).$$

(iv) Eine numerische Funktion heißt *primitiv-rekursive Funktion*, wenn sie durch Kompositionen und primitive Rekursionen aus den Grundfunktionen entsteht.

Satz 1.24 Jede primitiv-rekursive Funktion ist λ-definierbar. Insbesondere gelten:

(i) Die Grundfunktionen sind λ-definierbar vermöge
- Projektion $\lambda x_1 \ldots x_n.x_i$
- Nullfunktion $\lambda x_1 \ldots x_n.\mathsf{C}_0$
- Nachfolgerfunktion $\lambda nfx.f(nfx)$.

(ii) Falls g und $h_1, \ldots, h_n$ λ-definierbar sind, dann ist deren Komposition f λ-definierbar als

$$F = \lambda x_1 \ldots x_m.G(H_1 x_1 \ldots x_m) \ldots (H_n x_1 \ldots x_m).$$

(iii) Falls g und h λ-definierbar sind, dann ist deren primitive Rekursion f λ-definierbar als

$$F = \mathsf{Y}(\lambda F x y_1 \ldots y_n.\mathsf{ite}(\mathsf{iszero}x)(Hy_1 \ldots y_n)(G(F(\mathsf{pred}x)y_1 \ldots y_n)(\mathsf{pred}x)y_1 \ldots y_n)$$

Definition 1.25 (μ-Rekursion, μ-rekursive Funktion)

(i) Die *μ-Rekursion* $f = \mu g\colon \mathbb{N}^n \to \mathbb{N}$ einer numerischen Funktion $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$ ist definiert durch

$$f(x_1, \ldots, x_n) := \min\{y \mid g(x_1, \ldots, x_n, y) = 0\}.$$

(ii) Eine numerische Funktion heißt *μ-rekursive Funktion*, falls sie durch Kompositionen, primitive Rekursionen und μ-Rekursionen aus den Grundfunktionen entsteht.

Satz 1.26 Jede μ-rekursive Funktion ist λ-definierbar. Insbesondere gilt: Falls g λ-definierbar ist, dann ist deren μ-Rekursion f λ-definierbar als

$$F = H\mathsf{C}_0$$

$$H = \mathsf{Y}(\lambda H.\lambda y x_1 \dots x_n.\mathsf{ite}(\mathsf{iszero}(G x_1 \dots x_n y))y(H(\mathsf{succ}y)x_1 \dots x_n)).$$

Theorem 1.27 Die Turing-berechenbaren Funktionen entsprechen genau den μ-rekursiven Funktionen. Daher ist jede Turing-berechenbare Funktion auch λ-definierbar. Umgekehrt gilt sogar, dass jede λ-definierbare Funktion auch Turing-berechenbar ist.

1.1.7 Kombinatorische Logik

Definition 1.28 (CL-Term)

Für eine mindestens abzählbar unendliche Menge **V** von *Variablen* ist die Menge **CL** der **CL**-*Terme* über **V** induktiv definiert durch:

(i) $x \in \mathbf{V} \implies x \in \mathbf{CL}$

(ii) $\mathsf{K}, \mathsf{S} \in \mathbf{CL}$

(iii) $A, B \in \mathbf{CL} \implies (AB) \in \mathbf{CL}$ (*Applikation*)

Definition 1.29 (schwache Reduktion)

Auf den **CL**-Termen ist eine *schwache Reduktion* $\to_w$ definiert durch

(i) $\mathsf{K}AB \to_w A$

(ii) $\mathsf{S}ABC \to_w AC(BC)$

Analog zum λ-Kalkül wird die mehrfache schwache Reduktion durch $\Rightarrow_w$ gekennzeichnet. Jeder der in den drei Reduktionsregeln links stehenden **CL**-Terme heißt *schwacher Redex* und jeder der rechts stehenden **CL**-Terme heißt *schwaches Redukt*.

Definition 1.30 (schwache Normalform)

Ein **CL**-Term ist in *schwacher Normalform*, wenn er keine schwachen Redexe enthält.

Bemerkung 1.31 (i) Die Definitionen und Sätze aus dem Abschnitt 1.1.3 gelten entsprechend auch für **CL**-Terme und die schwache Re-

duktion. Insbesondere gelten auch das Church-Rosser-Theorem und das Normalisierungstheorem.

(ii) Die Reduktion $\to_w$ wird schwach genannt, weil es **CL**-Terme in schwacher Normalform gibt, deren transformierter λ-Term nicht in β-Normalform ist. Beispiel: $\mathsf{KI} = (\lambda xy.x)(\lambda x.x)$

Satz 1.32 (i) Transformation von **CL**-Termen in λ-Terme

$$
\begin{aligned}
(\cdot)_\lambda &: \mathbf{CL} \to \mathbf{\Lambda} \\
(x)_\lambda &= x \\
(AB)_\lambda &= (A)_\lambda (B)_\lambda \\
(\mathsf{K})_\lambda &= \lambda xy.x \\
(\mathsf{S})_\lambda &= \lambda xyz.xz(yz)
\end{aligned}
$$

(ii) Transformation von λ-Termen in **CL**-Terme

$$
\begin{aligned}
(\cdot)_{\mathbf{CL}} &: \mathbf{\Lambda} \to \mathbf{CL} \\
(x)_{\mathbf{CL}} &= x \\
(AB)_{\mathbf{CL}} &= (A)_{\mathbf{CL}} (B)_{\mathbf{CL}} \\
(\lambda x.A)_{\mathbf{CL}} &= \lambda^* x.(A)_{\mathbf{CL}}
\end{aligned}
$$

$$
\begin{aligned}
\lambda^* &: \mathbf{V} \times \mathbf{CL} \to \mathbf{CL} \\
\lambda^* x.x &= \mathsf{I} \\
\lambda^* x.A &= \mathsf{K}A \text{ für } A \neq x \\
\lambda^* x.AB &= \mathsf{S}(\lambda^* x.A)(\lambda^* x.B)
\end{aligned}
$$

Definition 1.33 (CL-Kombinatoren)

Die **CL**-Kombinatoren sind definiert als folgende λ-Terme:

- Identität: $\mathsf{I} = \mathsf{SKK} = \lambda x.x$
- Kanzellator: $\mathsf{K} = \lambda xy.x$
- Distributor: $\mathsf{S} = \lambda xyz.xz(yz)$
- Kompositor: $\mathsf{B} = \lambda xyz.x(yz)$
- Permutator: $\mathsf{C} = \lambda xyz.xzy$

Es gelten

$$\mathsf{S}(\mathsf{K}A)(\mathsf{K}B) = \mathsf{K}(AB)$$
$$\mathsf{S}(\mathsf{K}A)\mathsf{I} = A$$
$$\mathsf{S}(\mathsf{K}A)B = \mathsf{B}AB$$
$$\mathsf{S}A(\mathsf{K}B) = \mathsf{C}AB$$

1.2 Getypter Lambda-Kalkül

1.2.1 Monomorph getypter Lambda-Kalkül $\lambda\!\!\rightarrow$

implizite Typung (Curry)

$$\mathbf{T} := \mathbf{V_T} \mid (\mathbf{T}\!\rightarrow\!\mathbf{T})$$
$$\mathbf{\Lambda} := \mathbf{V} \mid (\lambda\mathbf{V}.\mathbf{\Lambda}) \mid (\mathbf{\Lambda\Lambda})$$

$$(\lambda x.A)B \rightarrow_\beta A[x := B]$$

Definition 1.34 (Typumgebung)

Eine rechts-eindeutige Relation

$$\Gamma \subseteq \mathbf{V} \times \mathbf{T}$$

heißt *Typumgebung*. Für ein Element (x, τ) einer Typumgebung schreiben wir auch

$$x{:}\tau.$$

Weiter setzen wir

$$\mathrm{dom}(\Gamma) := \Gamma\mathbf{T} = \{x \in \mathbf{V} \mid \exists\tau \in \mathbf{T}\colon\ x{:}\tau \in \Gamma\}$$

$$\mathrm{cod}(\Gamma) := \mathbf{V}\Gamma = \{\tau \in \mathbf{T} \mid \exists x \in \mathbf{V}\colon\ x{:}\tau \in \Gamma\}$$

sowie

$$\Gamma, x{:}\tau := \Gamma \cup \{x{:}\tau\}$$

und

$$\Gamma|_X := \Gamma \cap (X \times \mathbf{T}) = \{x{:}\tau \in \Gamma \mid x \in X\}$$

für $X \subseteq \mathbf{V}$.

Definition 1.35 (Typaussage, Wohltypung)

Eine *Typaussage* ist von der Form

$$\Gamma \vdash A{:}\tau$$

für eine Typumgebung Γ, einen λ-Term A und einen Typ τ. Für $\emptyset \vdash A{:}\tau$ schreiben wir einfach $\vdash A{:}\tau$. Ein λ-Term A heißt *wohlgetypt*, falls es eine Typumgebung Γ und einen Typ τ gibt, sodass $\Gamma \vdash A{:}\tau$ gilt. In diesem Fall heißt τ eine *Wohltypung* für A.

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau} \quad \text{(Axiom)}$$

$$\frac{\Gamma, x{:}\tau \vdash A{:}\sigma}{\Gamma \vdash (\lambda x.A){:}(\tau{\to}\sigma)} \quad (\to\text{-Introduktion})$$

$$\frac{\Gamma \vdash A{:}(\tau{\to}\sigma),\ \Gamma \vdash B{:}\tau}{\Gamma \vdash (AB){:}\sigma} \quad (\to\text{-Elimination})$$

Satz 1.36 (Basislemma)

(i) $\Gamma \vdash A{:}\tau,\ \Gamma \subseteq \Gamma' \implies \Gamma' \vdash A{:}\tau$

(ii) $\Gamma \vdash A{:}\tau \implies \mathbf{FV}(A) \subseteq \mathrm{dom}(\Gamma)$

(iii) $\Gamma \vdash A{:}\tau \implies \Gamma|_{\mathbf{FV}(A)} \vdash A{:}\tau$

Satz 1.37 (Generierungslemma)

(i) $\Gamma \vdash x{:}\tau \implies x{:}\tau \in \Gamma$

(ii) $\Gamma \vdash (\lambda x.A){:}\tau \implies \exists \sigma, \rho \in \mathbf{T}\colon \Gamma, x{:}\sigma \vdash A{:}\rho,\ \tau = (\sigma{\to}\rho)$

(iii) $\Gamma \vdash (AB){:}\tau \implies \exists \sigma \in \mathbf{T}\colon \Gamma \vdash A{:}(\sigma{\to}\tau),\ \Gamma \vdash B{:}\sigma$

Lemma 1.38 Für einen wohlgetypten λ-Term sind auch alle Teilterme wohlgetypt.

Satz 1.39 (Substitutionslemma)

(i) $\Gamma, x{:}\tau \vdash A{:}\sigma,\ y \notin \mathbf{FV}(A) \implies \Gamma, y{:}\tau \vdash (A[x := y]){:}\sigma$

(ii) $\Gamma \vdash (\lambda x.A){:}\tau,\ y \notin \mathbf{FV}(A) \implies \Gamma \vdash (\lambda y.(A[x := y])){:}\tau$

(iii) $\Gamma \vdash A{:}\tau \implies (\Gamma[\sigma := \rho]) \vdash A{:}(\tau[\sigma := \rho])$

(iv) $\Gamma, x{:}\tau \vdash A{:}\sigma,\ \Gamma \vdash B{:}\tau \implies \Gamma \vdash (A[x := B]){:}\sigma$

Theorem 1.40 (Typerhaltungstheorem) Es gilt

$$\Gamma \vdash A{:}\tau,\ A \Rightarrow_\beta B \implies \Gamma \vdash B{:}\tau.$$

Theorem 1.41 (Turing) Alle wohlgetypten λ-Terme sind stark β-normalisierend.

Definition 1.42 ($A{:}\tau$?, $A{:}$?, ?:τ)

(i) *Typprüfungsproblem* $A{:}\tau$?: Sei eine Typumgebung Γ, ein λ-Term A und ein Typ τ gegeben. Gilt $\Gamma \vdash A{:}\tau$?

(ii) *Typisierbarkeitsproblem* A :?: Sei eine Typumgebung Γ und ein λ-Term A gegeben. Gibt es einen Typ τ, sodass $\Gamma \vdash A{:}\tau$ gilt?

(iii) *Typbewohntheitsproblem* ?:τ: Sei eine Typumgebung Γ und ein Typ τ gegeben. Gibt es einen λ-Term A, sodass $\Gamma \vdash A{:}\tau$ gilt?

Satz 1.43 Die drei Probleme $A{:}\tau$?, A:? und ?:τ sind für den monomorph getypten λ-Kalkül $\lambda \to$ entscheidbar.

Der Fixpunktkombinator

$$\mathsf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

aus dem ungetypten λ-Kalkül ist im monomorph getypten λ-Kalkül $\lambda \to$ nicht wohlgetypt. Das liegt daran, dass die Selbstapplikation (xx) nicht wohlgetypt ist, denn wäre $\Gamma \vdash (xx){:}\tau$, so folgt mit dem Generationslemma die Existenz eines Typs σ mit $\Gamma \vdash x{:}(\sigma{\to}\tau)$ und $\Gamma \vdash x{:}\sigma$. Wegen der Rechtseindeutigkeit einer Typumgebung folgt also $\sigma = (\sigma{\to}\tau)$, aber dies ist kein gültiger Typ, weil er nicht endlich ist.
Damit wäre nun keine Rekursion im getypten λ-Kalkül möglich, und es wären viele Funktionen nicht definierbar. Als Ausweg führen wir für jeden Typ einen eigenen Fixpunktpunktkombinator als Konstante mit einer sogenannten δ-Konversionsregel ein.

Definition 1.44 (Fixpunktkombinator)

Für jeden Typ τ definieren wir einen *Fixpunktkombinator*

$$\mathsf{Y}_\tau{:}((\tau{\to}\tau){\to}\tau)$$

zusammen mit der δ-Konversionsregel

$$\mathsf{Y}_\tau F \to_\delta F(\mathsf{Y}_\tau F)$$

für beliebige λ-Terme $F{:}(\tau{\to}\tau)$.

explizite Typung (Church)

$$\mathbf{T} := \mathbf{V_T} \mid (\mathbf{T}{\to}\mathbf{T})$$
$$\mathbf{\Lambda} := \mathbf{V} \mid (\lambda\mathbf{V}{:}\mathbf{T}.\mathbf{\Lambda}) \mid (\mathbf{\Lambda\Lambda})$$

$$(\lambda x{:}\tau.A)B \to_\beta A[x := B]$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau} \quad \text{(Axiom)}$$

$$\frac{\Gamma, x{:}\tau \vdash A{:}\sigma}{\Gamma \vdash (\lambda x{:}\tau.A){:}(\tau{\to}\sigma)} \quad (\to\text{-Introduktion})$$

$$\frac{\Gamma \vdash A{:}(\tau{\to}\sigma),\ \Gamma \vdash B{:}\tau}{\Gamma \vdash (AB){:}\sigma} \quad (\to\text{-Elimination})$$

Satz 1.45 (Eindeutigkeitslemma)

(i) $\Gamma \vdash A{:}\tau,\ \Gamma \vdash A{:}\sigma \implies \tau = \sigma$

(ii) $\Gamma \vdash A{:}\tau,\ \Gamma \vdash B{:}\sigma,\ A =_\beta B \implies \tau = \sigma$

Definition 1.46 ()

Die Abbildung $|\cdot|\colon \mathbf{\Lambda}_{\text{exp}} \to \mathbf{\Lambda}_{\text{imp}}$ ordnet jedem explizit getypten λ-Term einen implizit getypten λ-Term zu, indem die Typinformationen der gebundenen Variablen entfernt werden.

$$|x| := x$$
$$|(\lambda x{:}\tau.A)| := (\lambda x.|A|)$$
$$|(AB)| := (|A||B|)$$

Satz 1.47 (i) Für alle $A \in \mathbf{\Lambda}_{\text{exp}}$ gilt $\Gamma \vdash A{:}\tau \implies \Gamma \vdash |A|{:}\tau$

(ii) Für alle $A \in \mathbf{\Lambda}_{\text{imp}}$ gilt $\Gamma \vdash A{:}\tau \implies \exists B \in \mathbf{\Lambda}_{\text{exp}}\colon A = |B|,\ \Gamma \vdash B{:}\tau$

1.2.2 Polymorph getypter Lambda-Kalkül $\lambda 2$

implizite Typung (Curry)

$$\mathbf{T} := \mathbf{V_T} \mid (\mathbf{T} \to \mathbf{T}) \mid (\forall \mathbf{V_T}.\mathbf{T})$$
$$\mathbf{\Lambda} := \mathbf{V} \mid (\lambda \mathbf{V}.\mathbf{\Lambda}) \mid (\mathbf{\Lambda}\mathbf{\Lambda})$$

$$(\lambda x.A)B \to_\beta A[x := B]$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau} \quad \text{(Axiom)}$$

$$\frac{\Gamma, x{:}\tau \vdash A{:}\sigma}{\Gamma \vdash (\lambda x.A){:}(\tau \to \sigma)} \quad (\to\text{-Introduktion})$$

$$\frac{\Gamma \vdash A{:}(\tau \to \sigma),\ \Gamma \vdash B{:}\tau}{\Gamma \vdash (AB){:}\sigma} \quad (\to\text{-Elimination})$$

$$\frac{\Gamma \vdash A{:}\tau,\ \sigma \notin \mathbf{FV}(\Gamma)}{\Gamma \vdash A{:}(\forall \sigma.\tau)} \quad (\forall\text{-Introduktion})$$

$$\frac{\Gamma \vdash A{:}(\forall \sigma.\tau)}{\Gamma \vdash A{:}(\tau[\sigma := \rho])} \quad (\forall\text{-Elimination})$$

Satz 1.48 Die drei Probleme $A{:}\tau?$, $A{:}?$ und $?{:}\tau$ sind für den polymorph implizit getypten λ-Kalkül zweiter Ordnung $\lambda 2$ nicht entscheidbar.

Satz 1.49 In polymorph getypten λ-Kalkül zweiter Ordnung $\lambda 2$ sind alle Funktionen λ-definierbar, die in Peano-Arithmetik zweiter Ordnung terminieren. Das sind unter anderem alle primitiv-rekursiven Funktionen.

explizite Typung (Church)

$$\mathbf{T} := \mathbf{V_T} \mid (\mathbf{T} \to \mathbf{T}) \mid (\forall \mathbf{V_T}.\mathbf{T})$$
$$\mathbf{\Lambda} := \mathbf{V} \mid (\lambda \mathbf{V}{:}\mathbf{T}.\mathbf{\Lambda}) \mid (\mathbf{\Lambda}\mathbf{\Lambda}) \mid (\Lambda \mathbf{V_T}.\mathbf{\Lambda}) \mid (\mathbf{\Lambda}\mathbf{T})$$

$$(\lambda x{:}\tau.A)B \to_\beta A[x := B]$$
$$(\Lambda \tau.A)\sigma \to_\beta A[\tau := \sigma]$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau} \quad \text{(Axiom)}$$

$$\frac{\Gamma, x{:}\tau \vdash A{:}\sigma}{\Gamma \vdash (\lambda x{:}\tau.A){:}(\tau{\to}\sigma)} \quad (\to\text{-Introduktion})$$

$$\frac{\Gamma \vdash A{:}(\tau{\to}\sigma),\ \Gamma \vdash B{:}\tau}{\Gamma \vdash (AB){:}\sigma} \quad (\to\text{-Elimination})$$

$$\frac{\Gamma \vdash A{:}\tau,\ \sigma \notin \mathbf{FV}(\Gamma)}{(\Lambda\sigma.A){:}(\forall\sigma.\tau)} \quad (\forall\text{-Introduktion})$$

$$\frac{\Gamma \vdash A{:}(\forall\sigma.\tau)}{\Gamma \vdash (A\rho){:}(\tau[\sigma := \rho])} \quad (\forall\text{-Elimination})$$

Satz 1.50 Die beiden Probleme $A{:}\tau$? und $A{:}$? sind für den polymorph explizit getypten λ-Kalkül zweiter Ordnung $\lambda 2$ entscheidbar. Das Problem ?$:\tau$ ist in $\lambda 2$ nicht entscheidbar.

1.2.3 Polymorph getypter Lambda-Kalkül $\lambda 3$

$$\mathbf{K} := * \mid (\mathbf{K}{\Rightarrow}\mathbf{K})$$

$$\mathbf{T} := \mathbf{V_T} \mid (\mathbf{T}{\to}\mathbf{T}) \mid (\forall\mathbf{V_T}{::}\mathbf{K}.\mathbf{T}) \mid (\lambda\mathbf{V_T}{::}\mathbf{K}.\mathbf{T}) \mid (\mathbf{TT})$$

$$\mathbf{\Lambda} := \mathbf{V} \mid (\lambda\mathbf{V}{:}\mathbf{T}.\mathbf{\Lambda}) \mid (\mathbf{\Lambda\Lambda}) \mid (\Lambda\mathbf{V_T}{::}\mathbf{K}.\mathbf{\Lambda}) \mid (\mathbf{\Lambda T})$$

$$(\lambda x{:}\tau.A)B \to_\beta A[x := B]$$

$$(\Lambda\tau{::}k.A)\sigma \to_\beta A[\tau := \sigma]$$

$$(\lambda\tau{::}k.\sigma)\rho \to_\beta \sigma[\tau := \rho]$$

$$\frac{\tau{::}k \in \Gamma}{\Gamma \vdash \tau{::}k} \quad (\text{Axiom 1})$$

$$\frac{\Gamma \vdash \tau{::}*,\ \Gamma \vdash \sigma{::}*}{\Gamma \vdash (\tau{\to}\sigma){::}*} \quad (\to\text{-}*\text{-Regel})$$

$$\frac{\Gamma, \sigma{::}k \vdash \tau{::}*,\ \sigma \notin \mathbf{FV}(\Gamma)}{\Gamma \vdash (\forall\sigma{::}k.\tau){::}*} \quad (\forall\text{-}*\text{-Regel})$$

$$\frac{\Gamma \vdash A{:}\tau,\ \Gamma \vdash \sigma{::}*,\ \tau =_\beta \sigma}{\Gamma \vdash A{:}\sigma} \quad (\beta\text{-}*\text{-Konvertibilität})$$

$$\frac{\Gamma, \tau{::}k \vdash \sigma{::}l}{\Gamma \vdash (\lambda\tau{::}k.\sigma){::}(k{\Rightarrow}l)} \quad (\Rightarrow\text{-Introduktion})$$

$$\frac{\Gamma \vdash \tau{::}(k{\Rightarrow}l),\ \Gamma \vdash \sigma{::}k}{\Gamma \vdash (\tau\sigma){::}l} \quad (\Rightarrow\text{-Elimination})$$

$$\frac{\Gamma \vdash \tau::*,\ x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau} \quad \text{(Axiom 2)}$$

$$\frac{\Gamma \vdash \tau::*,\ \Gamma, x{:}\tau \vdash A{:}\sigma}{\Gamma \vdash (\lambda x{:}\tau.A){:}(\tau{\to}\sigma)} \quad (\to\text{-Introduktion})$$

$$\frac{\Gamma \vdash A{:}(\tau{\to}\sigma),\ \Gamma \vdash B{:}\tau}{\Gamma \vdash (AB){:}\sigma} \quad (\to\text{-Elimination})$$

$$\frac{\Gamma, \sigma{::}k \vdash A{:}\tau,\ \sigma \notin \mathbf{FV}(\Gamma)}{\Gamma \vdash (\Lambda\sigma{::}k.A){:}(\forall\sigma{::}k.\tau)} \quad (\forall\text{-Introduktion})$$

$$\frac{\Gamma \vdash A{:}(\forall\sigma{::}k.\tau),\ \Gamma \vdash \rho{::}k}{\Gamma \vdash (A\rho){:}(\tau[\sigma := \rho])} \quad (\forall\text{-Elimination})$$

1.2.4 Curry-Howard-Isomorphismus

Theorem 1.51 (Curry-Howard-Isomorphismus)

(i) $\Gamma \vdash A{:}\tau \implies \mathrm{cod}(\Gamma) \vdash \tau$

(ii) $\Gamma' \vdash \tau \implies \exists A \in \mathbf{\Lambda}\colon \Gamma \vdash A{:}\tau$ mit $\Gamma = \{x_\tau{:}\tau \mid \tau \in \Gamma'\}$

2 Haskell

2.1 Grundprinzipien

2.1.1 referentielle Transparenz

Haskell ist *referentiell transparent*, d.h. ein Variable hat an allen Stellen ihres Geltungsbereich denselben Wert. Dies ermöglicht einen mathematischen Umgang mit Programmen, d.h. Programmeigenschaften können recht einfach bewiesen und Programme transformiert werden.

2.1.2 Auswertungsstrategien

Call-by-Value: LI-Reduktion, strike Auswertung, eager evaluation, application order reduction
Call-by-Name: LO-Reduktion, nicht-strikte Auswertung, normal order reduction
Call-by-Need: lazy evaluation; eine nicht-strikte Auswertung, bei der Ausdrücke, die zum Ergebnis beitragen und mehrmals auftreten, nur einmal ausgewertet werden

2.1.3 funktionale Programme

Ein *funktionales Programm* ist eine Folge von Funktionsdefinitionen.

```
f1 x11 ... x1m1 = A1

        ⋮

fn xn1 ... xnmn = An
```

Die *Auswertung* einer Funktionsdefinition ist die Belegung ihrer Variablen durch Werte.

```
f B1 ... Bn = A[x1:=B1,...,xn:=Bn]
```

Funktionen können auch rekursiv definiert sein.

2.1.4 Typen

Haskell ist eine *getypte Sprache* und verwendet das HINDLEY-MILNER-Typsystem. Es gibt *Basistypen* und *Typkonstruktoren*, wie $\rightarrow$ und $\times$, die zusammengesetzte Typen erzeugen.
Haskell ist *stark getypt*, d.h. zur Laufzeit können keine Fehler durch Anwendung von Funktionen auf Argumente des falschen Typs entstehen.
Haskell ist *statisch getypt*, d.h. zur Übersetzungszeit sind die Typen aller Ausdrücke bekannt, sie werden automatisch inferiert.

2.1.5 algebraische Datenstrukturen (abstrakte und rekursive Datentypen)

Eine *algebraische Datenstruktur* wird durch einen *Typkonstruktor* sowie endlich vielen *Datenkonstruktoren* definiert. Datenstrukturen können auch rekursiv definiert sein. Damit sie endlich sind, müssen jedoch auch nullstellige Datenkonstruktoren vorhanden sein. Zum Beispiel haben Listen den Typkonstruktor `[]`, den nullstelligen Datenkonstruktor `[]` und zweistelligen Datenkonstruktor `:`.

2.1.6 Pattern Matching

Funktionsdefinitionen von Funktionen auf algebraischen Datenstrukturen können intuitiv durch *Pattern Matching* geschrieben werden, indem für bestimmte Muster von Argumenten jeweils eine eigene Definition erfolgt, sodass alle möglichen Fälle abgedeckt sind. Geeignete Muster sind stets die verschiedenen Datenkonstruktoren eines Datentyps. Beispielsweise kann die Funktion zur Ermittlung der Länge einer Liste rekursiv definiert werden:

```
length [] = 0

length (h:t) = 1 + (length t)
```

2.1.7 Polymorphie

parametrische Polymorphie: Eine Funktion ist *parametrisch polymorph*, wenn sie auf verschiedenen Typen gleichartig wirkt. Zum Beispiel die Funktion `length`.
Ad-Hoc-Polymorphie: Eine Funktion ist *Ad-Hoc-polymorph*, wenn sie überladen ist, d.h. wenn sie auf verschiedenen Typen auch verschiedenartig wirkt. Dies tritt in Typklassen auf.

2.1.8 Typklassen

Eine *Typklasse* ist eine Menge von Typen, die durch die Namen und Typen der auf sie anwendbaren Operationen charakterisiert ist.

```
class <context> <class> <type> where <definitions>
```

Ein Element einer Typklasse heißt *Instanz* dieser Typklasse.

```
instance <context> <class> <type> where <definitions>
```

Die Namen der Operationen sind überladen, da sie in allen Instanzen der Typklasse gleich bezeichnet sind, aber unterschiedliche Definitionen haben können. Die Operationen sind also Ad-Hoc-polymorph. Die Typklassen sind hierarchisch geordnet.

2.1.9 Typinferenz

Sei V eine abzählbar unendliche Menge von Variablen und $\Theta = \bigcup_{n\in\mathbb{N}} \Theta_n$ eine Menge von Typkonstruktoren, wobei Θ_n die Menge der n-stelligen Typkonstruktoren ist. Die Menge Θ_0 entspricht der Menge der Basistypen. Die Menge $T = Typ(V,\Theta)$ der polymorphen Typen über V und Θ ist definiert durch

$$T ::= V \mid (T \to T) \mid (\Theta T \ldots T)$$

Eine Funktionsdefinition hat die Form

```
f x1 ... xn = A
```

und dabei sind die `xi` Variablen oder Datenkonstruktorapplikationen. Anhand dieser Gleichung kann der Typ von `f` inferiert werden.

(i) *Analyse der äußeren Struktur:* Es gilt

```
f :: a1 -> ... -> an -> a
```

Falls `xi` eine Datenkonstruktorapplikation ist, so kann der Typ `ai` durch eine entsprechende Typkonstruktorapplikation konkretisiert werden. Analog für `A` und `a`. Beispielsweise folgt aus `xi = []` oder `xi = (h:t)`, dass `ai = [b]` ein Listentyp ist.

(ii) *Analyse der inneren Struktur:* Für jede Applikation `BC` im Rumpf `A` wird eine Typgleichung der Form

```
typ(B) = typ(C) -> typ(BC)
```

aufgestellt. Bereits bekannte Typen werden sofort eingefügt, sonst werden stets neue Variablen verwendet.

(iii) *Unifikation:* Die Menge der Typgleichungen

$$\{\tau_i = \sigma_i \mid i \in I\}$$

wird unifiziert, d.h. es wird mit dem Unifikationsalgorithmus von ROBINSON eine Substitution

$$\phi\colon V \to Typ(V, \Theta)$$

berechnet, deren homomorphe Fortsetzung

$$\hat{\phi}\colon Typ(V, \Theta) \to Typ(V, \Theta)$$

die Menge der Typgleichungen löst, d.h. es gilt

$$\hat{\phi}(\tau_i) = \hat{\phi}(\sigma_i)$$

für alle $i \in I$. $\hat{\phi}$ heißt auch *Unifikator*. Die homomorphe Fortsetzung $\hat{\phi}$ ist definiert durch

$$\hat{\phi}(\tau) := \phi(\tau)$$

für alle Variablen $\tau \in V$,

$$\hat{\phi}(\tau{\to}\sigma) := (\hat{\phi}(\tau){\to}\hat{\phi}(\sigma))$$

für Funktionstypen und

$$\hat{\phi}(\theta\tau_1 \ldots \tau_n) := (\theta\hat{\phi}(\tau_1) \ldots \hat{\phi}(\tau_n))$$

für Strukturtypen. Nach dem Satz von ROBINSON existiert für eine Menge von Typgleichungen stets ein (bis auf Umbenennung von Typvariablen) eindeutiger allgemeinster Unifikator, den man durch die homomorphe Fortsetzung der Lösung des Unifikationsalgorithmus von ROBINSON erhält. Ein Unifikator ϕ heißt *allgemeinster Unifikator*, falls es für jeden weiteren Unifikator ψ eine Substitution

$$\rho\colon V \to Typ(V, \Theta)$$

mit

$$\psi = \hat{\rho} \circ \phi$$

gibt.

$$
\begin{aligned}
unify(\tau,\sigma) = & \mathit{if}\ (\tau \in V\ \wedge\ \tau \notin \sigma)\ \mathit{then}\ (\tau \mapsto \sigma)\\
& \mathit{elseif}\ (\sigma \in V\ \wedge\ \sigma \notin \tau)\ \mathit{then}\ (\sigma \mapsto \tau)\\
& \mathit{elseif}\ ((\tau \in \Theta_0\ \vee \sigma \in \Theta_0)\ \wedge\ \tau = \sigma)\ \mathit{then}\ id\\
& \mathit{elseif}\ (\tau = \tau_1 \to \tau_2\ \wedge \sigma = \sigma_1 \to \sigma_2)\\
& \quad \mathit{then}\ unifylist([\tau_1,\tau_2],[\sigma_1,\sigma_2],id)\\
& \mathit{elseif}\ (\tau = \theta\tau_1 \ldots \tau_n\ \wedge\ \sigma = \theta\sigma_1 \ldots \sigma_n)\\
& \quad \mathit{then}\ unifylist([\tau_1,\ldots,\tau_n],[\sigma_1,\ldots,\sigma_n],id)\\
& \mathit{else}\ fail
\end{aligned}
$$

$$
\begin{aligned}
unifylist([],[],\phi) = & \phi\\
unifylist(\tau : t, \sigma : s, \phi) = & let\ (\psi = unify(\phi(\tau),\phi(\sigma)))\\
& \quad in\ if\ (\psi = fail)\ then\ fail\ else\ unifylist(t,s,\psi \circ \phi)
\end{aligned}
$$

2.1.10 Komprehensionen

Eine Mengenkomprehension ist beispielsweise

$$M = \{n^2 \mid n \in \mathbb{N},\ n \mod 2 = 0\}$$

und eine entsprechende Listenkomprehension ist

```
l = [n^2|n <- [0..],n `mod` 2 == 0].
```

Allgemein sind Listenkomprehensionen von der Form

```
[A|q1,...,qn],
```

wobei die `qi`

(i) *Generatoren* `pattern <- list` mit `pattern :: a` und `list :: [a]`,

(ii) *Selektoren* `predicate :: Bool`, oder

(iii) *lokale Bindungen* `let expression`, die in nachfolgenden Generatoren und Selektoren gelten,

sein können. Es gelten

```
[A|p <- l] = map (\p -> A) l
[A|b] = if b then [A] else []
[A|let e] = let e in [A]
[A|q1,q2] = [[A|q2]|q1]
```

2.1.11 Funktionen höherer Ordnung und partielle Applikation

Eine Funktion, die eine andere Funktion als Argument oder Ergebnis hat, heißt *Funktion höherer Ordnung*. Eine Unterversorgung einer Funktion mit Argumenten heißt auch *partielle Applikation*.

2.1.12 strikte und nicht-strikte Funktionen

Eine nicht-terminierende Berechnung wird mit $\perp$ symbolisiert. Eine Funktion f heißt *strikt im i-ten Argument*, wenn ihre Berechnung genau dann nicht terminiert, wenn die Berechnung des i-ten Elements nicht terminiert, d.h. falls $f(x_1, \ldots, x_{i-1}, \perp, x_{i+1}, \ldots, x_n) = \perp$ für alle x_j gilt. Andernfalls heißt f *nicht-strikt im i-ten Argument*. f heißt *strikt*, wenn f strikt in allen Argumenten ist, andernfalls *nicht-strikt*.

2.1.13 Parser

Parser implementieren Erkennungsmechanismen für Sprachen. Sie stellen nach Eingabe eines Textes fest, ob dieser zur Sprache gehört oder nicht. Im Fall der Zugehörigkeit gibt der Parser eine Untergliederung des Textes in Struktureinheiten an. Parser können beispielsweise den Datentyp

```
type Parser tok a = [tok] -> [(a,[tok])]
```

verwenden.

2.1.14 Konstruktorklassen

Konstruktorausdrücke sind entweder Konstruktorvariablen, Typkonstruktoren oder Konstruktorapplikationen. Jeder Konstruktorausdruck hat einen Kind. Eine *Konstruktorklasse* ist eine Menge von Typkonstruktoren gleichen Kinds. Zum Beispiel die Klasse der Monaden oder der Funktoren.

```
class <context> <class> <type> where <definitions>

instance <context> <class> <type> where <definitions>
```

2.2 Verifikation

2.2.1 Wohlfundierte Induktion

Definition 2.1 (wohlfundierte Menge)

Eine geordnete Menge $(M, \leq)$ heißt *wohlfundiert*, falls jede nichtleere Teilmenge von M (mindestens) ein minimales Element besitzt.

Theorem 2.2 (wohlfundierte Induktion) Sei $(M, \leq)$ eine wohlfundierte Menge und P ein Prädikat über M. Dann gilt

$$\langle \forall x \in M\colon \{[\forall y < x\colon P(y)] \Longrightarrow P(x)\}\rangle \implies [\forall x \in M\colon P(x)].$$

Beweis: Für alle $x \in M$ gelte $[\forall y < x\colon P(y)] \Longrightarrow P(x)$. Sei

$$X := \{x \in M \mid \neg P(x)\},$$

angenommen es gäbe ein $x \in M$ mit $\neg P(x)$, d.h. $X \neq \emptyset$. Dann existiert wegen der Wohlfundiertheit ein minimales Element x_0 von X mit $\neg P(x_0)$. Weil x_0 ein minimales Element ist, folgt für alle $y < x_0$ stets $y \notin X$, also $P(y)$. Nach Voraussetzung ergibt sich damit aber $P(x_0)$. Widerspruch! ■

Möchte man also die Gültigkeit eines Prädikats P für alle Elemente einer wohlfundierten Menge $(M, \leq)$ zeigen, so geht man schrittweise vor:

(i) *Induktionsanfang:* Man zeigt, dass P für alle minimalen Elemente von M gilt.

(ii) *Induktionsschritt:* Unter der Voraussetzung, dass P für alle Elemente $y < x$ gilt, zeigt man dass P auch für x gilt.

Beispiel 2.3 Die vollständige Induktion ist eine wohlfundierte Induktion über der wohlfundierten Menge $(\mathbb{N}, \leq)$ der natürlichen Zahlen mit der üblichen Ordnung. □

Definition 2.4 (algebraischer Datentyp)

Ein *algebraischer Datentyp* ist definiert durch

$$D\tau_1 \ldots \tau_n = C_1\sigma_{11} \ldots \sigma_{1k_1} \mid \ldots \mid C_m\sigma_{m1} \ldots \sigma_{mk_m}$$

für $m > 0$, $k_i \in \mathbb{N}$ und $\sigma_{ij} \in T$ für $i \in \{0, \ldots, m\}$ und $j \in \{0, \ldots, k_i\}$. Dabei ist

$$T = T' \mid (D\tau_1 \ldots \tau_n)$$
$$T' = V \mid W \mid (T', \ldots, T') \mid (T' \to T')$$

und W ist die Menge aller konstanten Typen sowie aller algebraischen Datentypen. Die C_i heißen *Konstruktoren* und haben folgende Eigenschaften:

(i) Die Konstruktoren sind disjunkt, d.h. aus $C_i x_{i1} \dots x_{ik_i} = C_j y_{j1} \dots y_{jk_j}$ folgt $i = j$.

(ii) Die Konstruktoren sind injektiv, d.h. aus $C_i x_{i1} \dots x_{ik_i} = C_i y_{i1} \dots y_{ik_i}$ folgt $x_{ij} = y_{ij}$.

(iii) Die Konstruktoren sind surjektiv, d.h. für alle $x \in D\tau_1 \dots \tau_n$ gibt es ein i mit $x = C_i y_{i1} \dots y_{ik_i}$.

Beispiel 2.5 (i) Die strukturelle Induktion über einen algebraischen Datentyp $\mathcal{D}$ ist eine wohlfundierte Induktion über der wohlfundierten Menge $(\mathcal{D}, \leq)$. Die minimalen Elemente von $(\mathcal{D}, \leq)$ sind die Elemente der Konstruktoren, die $\mathcal{D}$ nicht beinhalten. Die Elemente der Konstruktoren, die $\mathcal{D}$ beinhalten, sind echt größer als die Elemente von $\mathcal{D}$, aus denen sie konstruiert werden.

(ii) Die Listeninduktion ist eine strukturelle Induktion über den algebraischen Datentyp

$$\mathcal{L} = \text{List}\tau = \text{Empty} \mid \text{Cons}\tau(\text{List}\tau).$$

Das einzige minimale Element ist hier Empty und es gilt $L < \text{Cons}xL$ für alle $x \in \tau$ und $L \in \mathcal{L}$.

(iii) Die Bauminduktion ist eine strukturelle Induktion über den algebraischen Datentyp

$$\mathcal{T} = \text{Tree}\tau = \text{Leaf}\tau \mid \text{Node}\tau(\text{Tree}\tau)(\text{Tree}\tau).$$

Die minimalen Elemente sind Leaf$x$ für alle $x \in \tau$. Weiterhin gilt $T_1 < \text{Node}xT_1T_2$ und $T_2 < \text{Node}xT_1T_2$ für alle $x \in \tau$ und $T_1, T_2 \in \mathcal{T}$. □

2.3 Transformation

2.3.1 Fold-Unfold-Methode

Definition 2.6 (Fold-Unfold-Methode)

(i) *Definition:* Eine neue Funktionsgleichung wird eingefügt, deren linke Seite keine Spezialisierung einer bereits existierenden Funktionsgleichung sein darf.

(ii) *Instanz:* Eine spezialisierte Funktionsgleichung wird eingefügt, indem in einer bereits existierenden Funktionsgleichung für Variablen Ausdrücke einsetzt.

(iii) *Unfold:* Eine neue Funktionsgleichung wird eingefügt, indem in einer bereits existierenden Funktionsgleichung ein Funktionsaufruf durch den definierenden Ausdruck ersetzt wird.

(iv) *Fold:* Eine neue Funktionsgleichung wird eingefügt, indem in einer bereits existierenden Funktionsgleichung ein Ausdruck durch einen Funktionsaufruf ersetzt wird.

(v) *Abstraktion:* Eine neue Funktionsgleichung wird eingefügt, indem in einer bereits existierenden Funktionsgleichung lokale Definitionen eingeführt werden.

(vi) *Gesetze:* Eine neue Funktionsgleichung wird eingefügt, indem (algebraische) Gesetze auf die rechte Seite einer bereits existierenden Funktionsgleichung angewendet werden.

Beispiel 2.7 Für eine Liste ganzer Zahlen soll der Mittelwert berechnet werden.

```
avg l = (sum l) `div` (length l)
```

Unfold

```
avg l = (foldr (+) 0 l) `div` (sum (map (const 1) l))
```

Unfold

```
avg l = (foldr (+) 0 l) `div` (foldr (+) 0 (map (const 1) l))
```

Abstraktion

```
avg l = x `div` y
```

```
 where (x,y) = (foldr (+) 0 l,foldr (+) 0 (map (const 1) l))
```

Definition

```
accum l = (foldr (+) 0 l,foldr (+) 0 (map (const 1) l))
```

Gesetze

```
accum l = (foldr (+) 0 l,foldr (⊕) 0 l)
            where x⊕n = 1+n
```

Gesetze

```
accum l = foldr (⊞) (0,0) l
 where x⊞(m,n) = (x+m,1+n)
```

□

2.3.2 Bird-Meertens-Formalismus

Theorem 2.8 (Dualitätssatz)

(i) *Erster Dualitätssatz:*

```
foldl f e = foldr f e
```

falls `f` eine assoziative Operation mit neutralem Element `e` ist.

(ii) *Zweiter Dualitätssatz:*

```
foldl f e = (foldr (flip f) e).reverse
```

(Beachte: Es gilt `flip.flip = id` und `reverse.reverse = id`.)

Beispiel 2.9 Wegen `id = foldr (:) []` folgt aus dem zweiten Dualitätssatz

```
reverse = id.reverse = (foldr (:) []).reverse = foldl (flip (:)) []
```

□

Satz 2.10 *Fold-Fusion:*

```
f.(foldl g e) = foldl h (f e)
```

für `f.g x y = h.f x y` und

```
f.(foldr g e) = foldr h (f e)
```

für `f.g x y = (flip ((flip h).f)) x y`

Theorem 2.11 (Bird-Meertens-Formalismus)

(i) *Map-Distributivität:*

```
(map f).(map g) = map (f.g)
```

(ii) *Scan-Lemma:*

```
(map (foldl f e)).inits = scanl f e

(map (foldr f e)).tails = scanr f e
```

(iii) *Map-Promotion:*

```
(map f).concat = concat.(map (map f))
```

(iv) *Fold-Promotion:*

```
(foldl ⋉ e).concat = (foldl ⋉ e).(map (foldl ⋉ e))
```

falls $\ltimes$ eine assoziative Operation mit linksneutralem Element `e` und

```
(foldr ⋊ e).concat = (foldr ⋊ e).(map (foldr ⋊ e))
```

falls $\rtimes$ eine assoziative Operation mit rechtsneutralem Element `e` ist.

(v) *Fold-Map-Fusion:*

```
(foldl f e).(map g) = foldl (flip ((flip f).g)) e

(foldr f e).(map g) = foldr (f.g) e
```

(vi) *Fold-Scan-Fusion:*

```
(foldl ⊕ 0).(scanl ⊙ 1) = <<.(foldl ⊗ (0 ⊕ 1,1))
```

mit `x << y = x` und `(x,y) ⊗ z = (x ⊕ (y ⊙ z),y ⊙ z)`

(vii) *Fold-Fold-Fusion (Horner):*

```
(foldl ⊕ 0).(map (foldl ⊙ 1)).tails = foldl ⊗ 1
```

mit `x ⊗ y = (x ⊙ y) ⊕ 1` falls $\odot$ rechtsdistributiv über $\oplus$ und `0` linksneutrales Element von $\oplus$ ist.

2.4 Monaden

Definition 2.12 (Monade)

Eine *Monade* ist ein Typkonstruktor `M` zusammen mit zwei polymorphen Funktionen

```
unit :: a -> M a
```

```
(*) :: M a -> (a -> M b) -> M b
```

sodass folgende drei Gleichungen gelten:

(i)

```
m * unit = m
```

(ii)

```
(unit a) * f = f a
```

(iii)

```
(m * f) * g = m * (f ⊛ g)
```

Für einen Ausdruck der Form

```
m * (\a -> n)
```

schreiben wir äquivalent auch

```
let a = m in n
```

Satz 2.13 Für die Operation

```
(⊛) :: (a -> M b) -> (b -> M c) -> a -> M c

(f ⊛ g) a = (f a) ∗ g
```

gelten die folgenden drei Gesetze:

(i)
```
f ⊛ unit = f
```

(ii)
```
unit ⊛ f = f
```

(iii)
```
(f ⊛ g) ⊛ h = f ⊛ (g ⊛ h)
```

und es gilt
```
m ∗ f = (id ⊛ f) m
```

Beispiel 2.14 (i) *Identitätsmonade:*

```
type M a = a

unit a = a

a ∗ f = f a

f ⊛ g = g.f

map f a = f a

join a = a
```

(ii) *Listenmonade:*

```
type M a = [a]

unit a = [a]

[] ∗ f = []

(h:t) ∗ f = (f h) ++ (t ∗ f)

m ∗ f = concat.map f m

f ⊛ g = concat.(map g).f

map = map

join = concat
```

(iii) *Ausgabemonade:*

```
type M a = (a,String)

unit a = (a,‘‘ ‘‘)

m ∗ f = (b,s ++ s’) where { m = (a,s), f a = (b,s’) }

(f ⊛ g) a = (c,s ++ s’) where { f a = (b,s), g b = (c,s’) }

map f m = (f a,s) where m = (a,s)

join m = (a,s ++ s’) where m = ((a,s’),s)
```

(iv) *Zustandsmonade:*

```
type M a = State -> (a,State)

unit a = \s -> (a,s)

m ∗ f = \s -> f a s’ where m s = (a,s’)

(f ⊛ g) a = \s -> g b s’ where f a s = (b,s’)

map f m = \s -> (f a,s’) where m s = (a,s’)

join m = \s -> m’ s’ where m s = (m’,s’)
```

□

Satz 2.15 Für die Operationen

```
map :: (a -> b) -> M a -> M b
map f m = m * (unit.f)
```

und

```
join :: M (M a) -> M a
join m = m * id
```

gelten die folgenden sieben Gesetze:

(i)
```
map id = id
```

(ii)
```
(map f).(map g) = map (f.g)
```

(iii)
```
(map f).unit = unit.f
```

(iv)
```
(map f).join = join.(map (map f))
```

(v)
```
join.unit = id
```

(vi)
```
join.(map unit) = id
```

(vii)
```
join.(map join) = join.join
```

Theorem 2.16 Ein Typkonstruktor `M` ist genau dann eine Monade, wenn es Operationen `unit`, `map` und `join` gibt, die die sieben Gesetze aus dem vorigen Satz erfüllen. Weiterhin gilt

```
x * f = join.map f x
```

Literaturverzeichnis

[Bar84] BARENDREGT, H. P.: *The Lambda Calculus: Its Syntax and Semantics (Studies in Logic and the Foundations of Mathematics)*. Revised Edition. Elsevier Science, 1984. – ISBN 0444875082

[Bar91] BARENDREGT, H. P.: *Lambda Calculi with Types*. 1991

Index