Typprüfung für System F_ω mit CASE und PAIR

Markus Bleicher

12. Mai 2003

Dieses Fortgeschrittenenpraktikum war der Versuch, einen Typprüfungsalgorithmus für den auf Typkonstruktoren von höchstens Rang 3 beschränkten polymorphen λ-Kalkül System F_ω zu entwerfen und zu implementieren, der zur Verifikation von in Beweisen vorkommenden Typisierungen dienen soll.

Der dargestellte Algorithmus geht kanonisch aus dem syntaxgeleiteten Kalkül hervor. Es wird ein mit Typen annotierter Termbaum aufgebaut und in einen Ableitungsbaum des syntaxgeleiteten Kalküls überführt.

Die Implementierung beschränkt sich auf die Verifikation von Churchtermen.

Inhaltsverzeichnis

> "Rank 2 types break all the standard typing algorithms. We can only do type checking by making rather draconian restrictions." [1]

0.1 Das Problem

fasst obiges Zitat gut zusammenfassen.

Die Fragestellung dieser Arbeit entwickelte sich aus der Verifikation von Beweistermen für „Monotoniezeugen" in einer Erweiterung von System F_ω ([7, 8, 9]). Für bestimmte Typen sollen Terme gefunden oder verifiziert werden, die von diesen Typen getypt werden.

Da die Typgewinnung nicht nur in System F_ω sondern bereits in System F unentscheidbar ist, ausserdem Urzyczyn[2] mutmaßt, alle in System F_ω typisierbaren Terme seien bereits in System $F_\omega^{(1)}$ typisierbar und Typprüfung entgegen dem Anschein keinesfalls leichter als Typgewinnung ist, ist ein weites Feld für unfruchtbare Basteleien geöffnet.

Daher betrachte ich auch die entscheidbare Verifikation des Typens von Church-Termen oder die (in System F_ω unentscheidbare, in System $F_\omega^{(3)}$ entscheidbare) Typprüfung für vollständig getypte (annotierte) Curry-Terme.

[1] Simon Peyton Jones auf der Glasgow-Haskell-Bugs-Mailinglist
http://haskell.cs.yale.edu/pipermail/glasgow-haskell-bugs/2001-April/001394.html
[2] in [12], S.338 Conjecture 2.3

0.2 Typen und Typsysteme

(Einordnung von System F_ω. Für das Thema und Verständnis der Arbeit nebensächlich.)

Typen kann man als Prädikate für die Terme des ungetypten λ-Kalküls auffassen.

Es gibt heute eine Vielzahl verschiedener Typsysteme, die man zuerst einmal in zwei Klassen einteilen kann: Systeme, bei denen die Typen in den Termen enthalten sind („endogen", „explizit", „a la Church"), die Herleitungsregeln für syntaktisch korrekte Terme also bereits auf Typen Bezug nehmen. Und Systeme, bei denen die Typen nachträglich den Termen zugewiesen werden („exogen", „implizit", „a la Curry") (nach [10]).

In [10] werden die Typsysteme, die in [1] an Hand der drei Kriterien Vorhandensein von Polymorphie, abhängigen Typen und Typen höherer Ordnung klassifiziert, auf einem Würfel angeordnet .

Zwischen den beiden Würfeln für explizite und implizite Typsysteme, gibt es - wie in [10] gezeigt - eine Isomorphie für die eine Hälfte ohne abhängige Typen mit Hilfe einer Typlöschungs- und einer Typgewinnungsfunktion. Für die andere Hälfte mit abhängigen Typen existiert natürlicherweise nur eine Injektion.

Es folgen nun noch Teile der Definition eines umfassenden Typsystems aus [10], S. 90, in seiner endogenen Darstellung.

0.2.1 Sprache

Seien Var die Menge der Konstruktorvariablen (mit A, B als Mitteilungszeichen), var die Menge der Termvariablen (mit x,y als Mitteilungszeichen) zwei disjunkte Variablenmengen. Getypte Terme (bezeichnet durch m, n, r, s, t), Typ-Konstruktoren (bezeichnet durch kleine griechische Buchstaben) und Kinds (bezeichnet durch K, K') sind durch folgende Grammatik definiert:

$$\begin{array}{rcl} m & ::= & x|\lambda x:\phi.m|mm|\lambda A:K.m|m\phi \\ \phi & ::= & A|\Pi x:\phi.\phi|\lambda x:\phi.\phi|\Pi A:K.\phi|\lambda A:K.\phi|\phi\phi|\phi m \\ K & ::= & *|\Pi x:\phi.K|\Pi A:K.K \end{array}$$

Die β-Reduktion ist definiert als der reflexive und transitive Abschluss der Relation $(\lambda v:P.Q)R \to_\beta Q[v:=R]$, .

Die β-Gleichheit $=_\beta$ ist die durch β-Reduzierbarkeit und syntaktische Gleichheit unter α-Konversion bestimmte Kongruenzrelation.

0.2.2 Das General Type System Kalkül

Darstellung der Kalküle von Barendregt mit Hilfe des General Type System $\vdash_t$ im Church style:

Regeln für Kinds

$$\frac{}{\vdash_t * : \#}(\mathrm{AX})$$

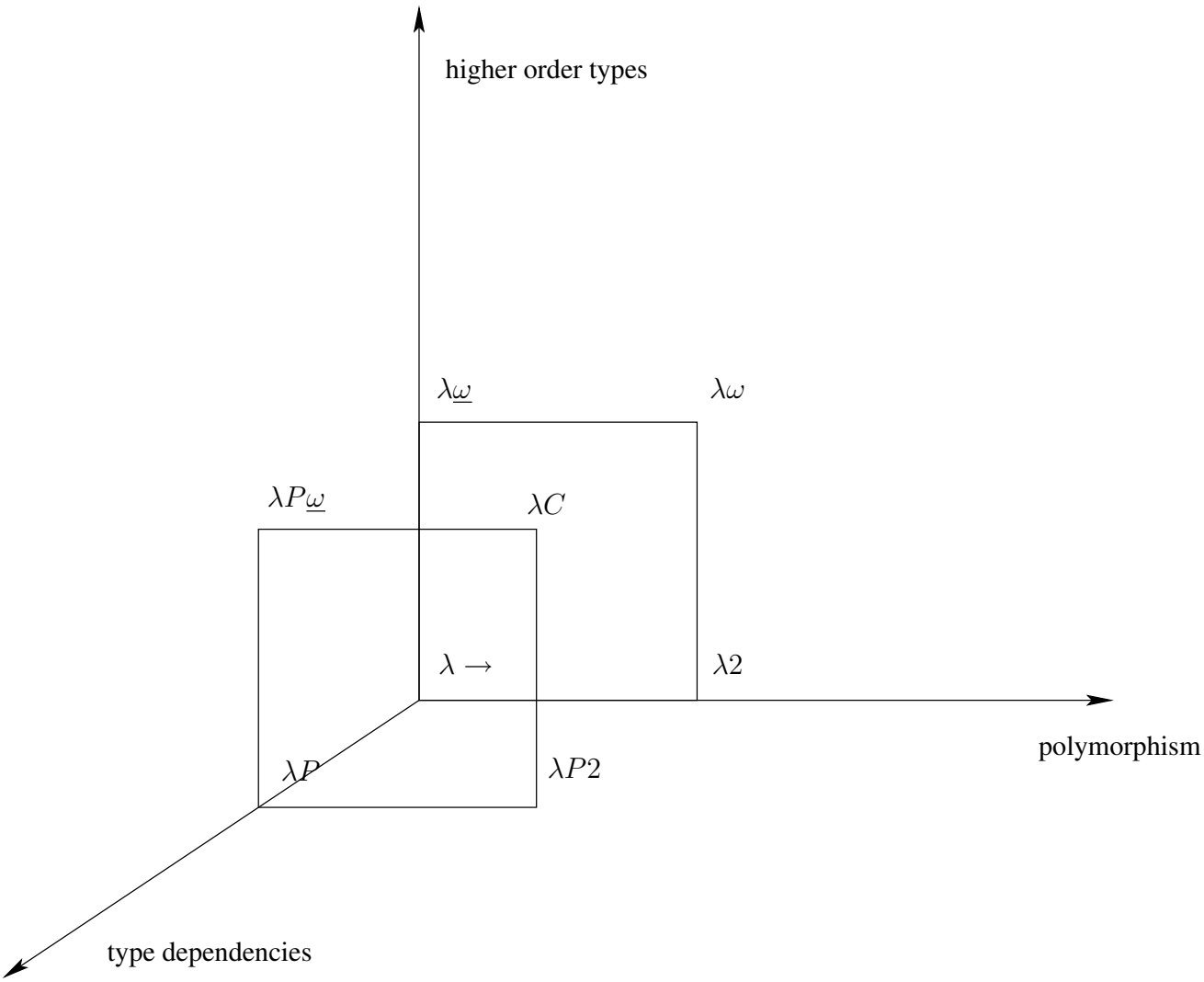

Abbildung 0.1: Barendegts Würfel mit endogen getypten λ-Kalkülen

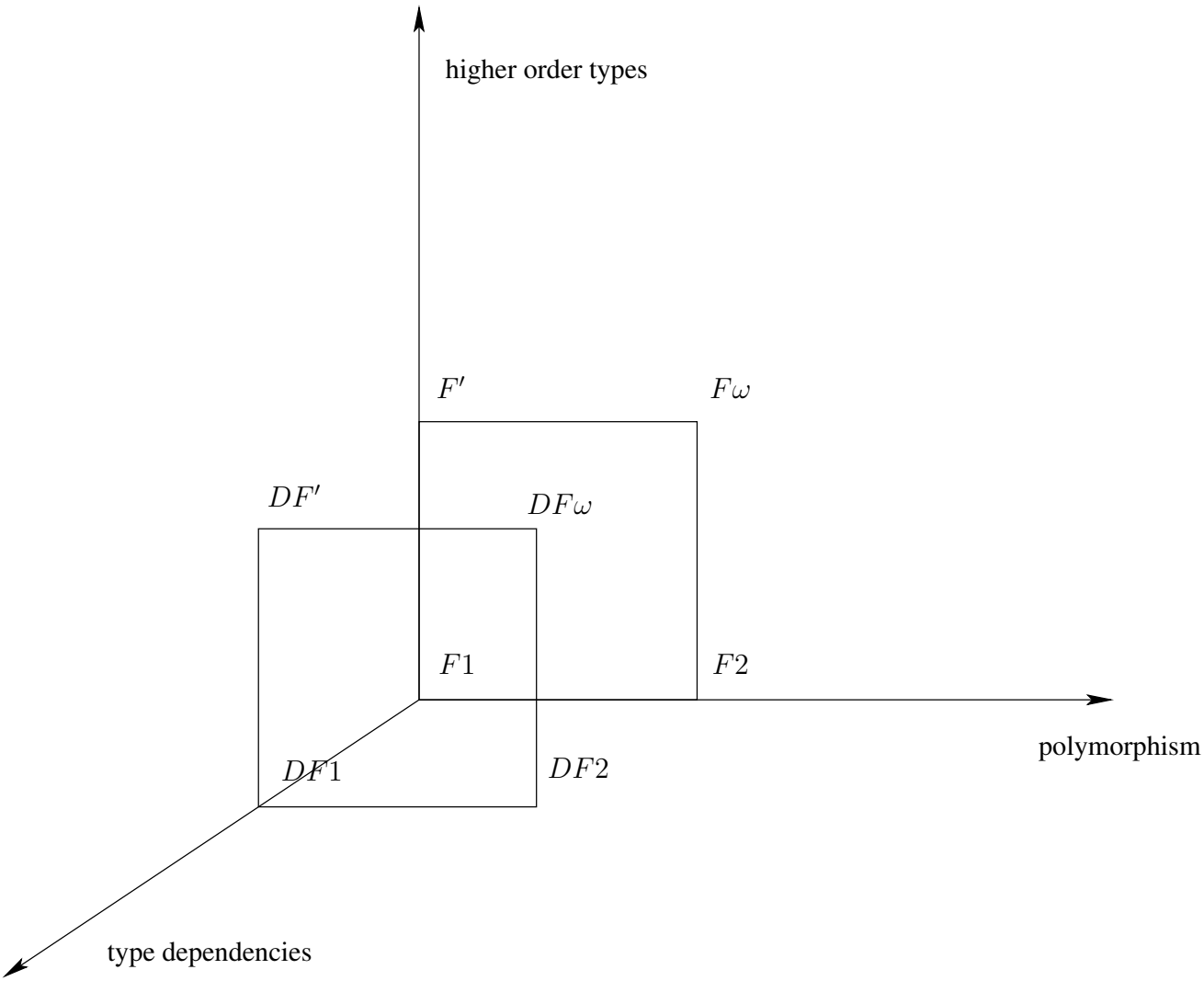

Abbildung 0.2: Barendegts Würfel mit exogenen Typsystemen

$$\frac{\Gamma, x : \rho \vdash_t K : \#}{\Gamma \vdash_t \Pi x : \rho . K : \#}(\text{K}\Pi\text{FC})$$

$$\frac{\Gamma, x : K \vdash_t K' : \#}{\Gamma \vdash_t \Pi x : K . K' : \#}(\text{K}\Pi\text{FK})$$

Regeln für Konstruktoren

$$\frac{\Gamma \vdash_t K : \#}{\Gamma, a : K \vdash_t a : K} a \notin \text{dom}(\Gamma)(\text{CVAR})$$

$$\frac{\Gamma \vdash_t \rho : K \quad \Gamma \vdash_t K' : \# \quad K =_\beta K'}{\Gamma \vdash_t \rho : K'}(\text{C} =_\beta)$$

$$\frac{\Gamma, x : \rho \vdash_t \psi : *}{\Gamma \vdash_t \Pi x : \rho . \psi : *}(\text{C}\Pi\text{FC})$$

$$\frac{\Gamma, a : K \vdash_t \psi : *}{\Gamma \vdash_t \Pi a : K . \psi : *}(\text{C}\Pi\text{FK})$$

$$\frac{\Gamma, x : \rho \vdash_t \psi : K}{\Gamma \vdash_t \lambda x : \rho . \psi : \Pi x : \rho . K}(\text{C}\Pi\text{IC})$$

$$\frac{\Gamma \vdash_t \sigma : \Pi x : \psi . K \quad \Gamma \vdash_t M : \psi}{\Gamma \vdash_t \sigma M : K[x := M]}(\text{C}\Pi\text{EC})$$

$$\frac{\Gamma, a : K \vdash_t \psi : K'}{\Gamma \vdash_t \lambda a : K . \psi : \Pi a : K . K'}(\text{C}\Pi\text{IK})$$

$$\frac{\Gamma \vdash_t \sigma : \Pi a : K . K' \quad \Gamma \vdash_t \psi : K}{\Gamma \vdash_t \sigma \psi : K'[a := \psi]}(\text{C}\Pi\text{EK})$$

Regeln für Terme

$$\frac{\Gamma \vdash \sigma : *}{\Gamma, x : \sigma \vdash x : \sigma} x \notin \text{dom}(\Gamma)(\text{VAR})$$

$$\frac{\Gamma \vdash_t M : \rho \quad \Gamma \vdash_t \psi : * \quad \rho =_\beta \psi}{\Gamma \vdash_t M : \psi}(=_\beta)$$

$$\frac{\Gamma, x : \rho \vdash_t M : \psi}{\Gamma \vdash_t \lambda x : \rho . M : \Pi x : \rho . \psi}(\Pi\text{IC})$$

$$\frac{\Gamma \vdash_t M : \Pi x : \phi . \psi \quad \Gamma \vdash_t N : \phi}{\Gamma \vdash_t MN : \psi[x := N]}(\Pi\text{EC})$$

$$\frac{\Gamma, a : K \vdash_t M : \psi}{\Gamma \vdash_t \lambda a : K . M : \Pi a : K . \psi}(\Pi\text{IK})$$

$$\frac{\Gamma \vdash_t M : \Pi a : K . \sigma \quad \Gamma \vdash_t \psi : K}{\Gamma \vdash_t M\psi : \sigma[a := \psi]}(\Pi\text{EK})$$

Und die Abschwächungsregel

$$\frac{\Gamma \vdash_t P : Q \quad \Gamma \vdash_t R : s}{\Gamma, u : R \vdash_t P : Q} u \notin \mathrm{dom}(\Gamma)(\mathrm{WEAK})$$

K ist ein wohlgeformter Kind in Γ, falls $\Gamma \vdash_t K : \#$. ϕ ist ein wohlgeformter Konstruktor in Γ vom Kind K, falls $\Gamma \vdash_t \phi : K$. Schliesslich ist M ein wohlgeformter Term vom Typ ϕ in Γ, falls $\Gamma \vdash_t M : \phi$.

0.2.3 Kalküle

	Name des Kalküls
$\lambda^{\rightarrow}$	einfach getypter Lamda-Kalkül
$\lambda 2$	Sytem F
$\lambda\underline{\omega}$	
$\lambda\omega$	Sytem F_ω
λP	
$\lambda P2$	
$\lambda P\underline{\omega}$	
λC	Calculus of Constructions

1 System F_ω und Erweiterungen

1.1 System F_ω

Das System F_ω ist der Lambda-Kalkül, dessen Typsystem Konstruktorvariablen beliebiger Ordnung enthält. Er ist das stärkste System ohne abhängige Typen. Da abhängige Typen die Menge der typbaren Terme nicht erhöhen ([10], S.99), ist es auch mit das stärkste Typsystem auf Barendregts Würfel. Ich verwende den Begriff System F_ω unabhängig davon, ob ein endogenes (das eigentliche System F_ω) Typsystem oder seine exogene Modifikation gemeint ist.

1.1.1 Sprache und Notation

Um den Kalkül in der üblichen Notation zu schreiben, identifiziert man $\Pi x : \rho.\psi$ mit $\rho \to \psi$ (da x in ψ nicht vorkommt) und $\Pi a : K.\psi$ mit $\forall a : K.\psi$.

Das Kindsystem - also das Typsystem der Typen von System F_ω - hat die Struktur eines einfach getypten Lambda-Kalküls. Dabei werden die Konstruktoren von Kinds der Form

$$K ::= *|(K \Rightarrow K)$$

getypt. Damit haben die Konstruktoren entweder die Form

$$\phi^{K \Rightarrow K'} ::= A^{K \Rightarrow K'}|\lambda A^K.\phi^{K'}|(\phi^{K'' \Rightarrow K \Rightarrow K'}\phi^{K''})$$

oder, falls sie von Kind * und damit Typen für die Terme von System F_ω sind, sind sie von der Gestalt

$$\phi^* ::= A^*|(\phi^* \to \phi^*)|\forall a^K.\psi^*|(\phi^{K_1 \Rightarrow K_2 \Rightarrow \ldots \Rightarrow K_n \Rightarrow *}\phi^{K_1 \Rightarrow K_2 \Rightarrow \ldots \Rightarrow K_n})$$

Terme sind

$$t ::= x|\lambda x t|(tt)$$

Zur Klammerersparnis seien die Applikation links-, die Pfeile rechtsassoziativ.

Die Substitution $t[x := s]$ ersetzt alle freien Vorkommnisse von x in t durch s. Dabei werden gebundene Variablen in t so umbenannt (α-Konversion), dass keine freien Variablen aus s in den Bindungsbereich eines Variablenbinders kommen, also „gefangen" werden.

Die parallele Substitution $t[x_1 := s_1, \ldots, x_n := s_n]$ ersetzt gleichzeitig alle Vorkommnisse von x_1 bis x_n. Mit $\overrightarrow{x} = x_1, \ldots, x_n$ und $\overrightarrow{s} = s_1, \ldots, s_n$ schreibe ich dafür auch $t[\overrightarrow{x} := \overrightarrow{s}]$. Genauso gilt: $\forall \overrightarrow{x} t = \forall x_1 \ldots \forall x_n t$ und $\lambda \overrightarrow{x} t = \lambda x_1 \ldots \lambda x_n t$. Die Vektoren können auch leer sein, die Definitonen meinen dann jeweils t.

Analoge Definition der Substitution und von Vektoren für Typen.

Die β-Reduktion der Typen ist stark normalisierend, da Typen mit Kinds einen einfach getypten Lambdakalkül bilden. β-gleiche Typkonstruktoren werden identifiziert.

1.1.2 Der Kalkül von System $\mathbf{F}_\omega$

Γ ist eine Variablenumgebung, also eine endliche Menge von getypten Termvariablen, wobei die Schreibweise $\Gamma(x)$ den Typ der Variablen x zurückgibt und das Komma eine getypte Variable einfügt.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}(\text{VAR})$$

$$\frac{\Gamma \vdash r : \rho \to \sigma \quad \Gamma \vdash s : \rho}{\Gamma \vdash rs : \sigma}(\text{APP})$$

$$\frac{\Gamma, x : \rho \vdash t : \sigma}{\Gamma \vdash \lambda x t : \rho \to \sigma}(\text{ABS})$$

$$\frac{\Gamma \vdash t : \forall X \sigma}{\Gamma \vdash t : \sigma[X := \rho]}(\text{INST})$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash t : \forall X \sigma}\text{if X not free in } \Gamma(\text{GEN})$$

Das System besitzt die subject reduction Eigenschaft, wenn also s zu einem s' reduziert und $\Gamma \vdash s : \sigma$ gilt, dann gilt auch $\Gamma \vdash s' : \sigma$.

1.1.3 Terme für $\mathbf{F}_\omega$ im Church style

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}(\text{VAR})$$

$$\frac{\Gamma \vdash r : \rho \to \sigma \quad \Gamma \vdash s : \rho}{\Gamma \vdash rs : \sigma}(\text{APP})$$

$$\frac{\Gamma, x : \rho \vdash t : \sigma}{\Gamma \vdash \lambda x : \rho . t : \rho \to \sigma}(\text{ABS})$$

$$\frac{\Gamma \vdash t : \forall X \sigma}{\Gamma \vdash t\rho : \sigma[X := \rho]}(\text{INST})$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \Lambda X t : \forall X \sigma}\text{if X not free in } \Gamma(\text{GEN})$$

1.1.4 Syntaxgeleitete Version des Kalküls

Aus [10]. Wenn man die Generalisierungen mit der davor liegenden Abstraktion, Applikation oder VAR-Anwendung zusammenzieht, Spezialisierungen immer mit der vorhergegangenen Applikation oder VAR-Anwendung ausführt, benötigt man nur noch 3 Regeln.

Definition der Matching-Relation $\leq$: Für zwei Typen ϕ und ψ gilt $\phi \leq \psi$ (ψ ist eine allquantifizierte Spezialisierung von ϕ) genau dann, wenn $\phi = \forall\overrightarrow{A}.\phi'$ und $\psi = \forall\overrightarrow{B}.\phi'[\overrightarrow{A} := \overrightarrow{\chi}]$, wobei die $\overrightarrow{B}$ nicht frei in $\phi = \forall\overrightarrow{A}.\phi'$ vorkommen dürfen (Henglein andersrum).

$$\frac{\Gamma(x) \leq \alpha}{\Gamma \vdash x : \alpha}(\mathrm{VAR}')$$

$$\frac{\Gamma \vdash r : \rho \to \sigma \quad \Gamma \vdash s : \rho \quad \sigma \leq \alpha}{\Gamma \vdash rs : \forall\overrightarrow{X}.\alpha}\overrightarrow{X} \text{ not free in } \Gamma(\mathrm{APP}')$$

$$\frac{\Gamma, x : \rho \vdash t : \sigma}{\Gamma \vdash \lambda x t : \forall\overrightarrow{X}.\rho \to \sigma}\overrightarrow{X} \text{ not free in } \Gamma(\mathrm{ABS}')$$

Der resultierende Kalkül ist syntaxgeleitet.

1.1.5 Äquivalenzbeweis

Die Äquivalenz der syntaxgeleiteten Version mit der ursprünglichen Variante wird in [10] auf Seite 104, Theorem 2.3.4 bewiesen.

1.1.6 Notation: Rang und $\mathrm{F}_\omega^{(n)}$

Der Rang o eines Kinds ist induktiv definiert: $\mathrm{o}(*) = 0$, $\mathrm{o}(k_1 \Rightarrow k_2) = \max(\mathrm{o}(k_1) + 1, \mathrm{o}(k_1))$. Da jeder Kind in der Form $k_1 \Rightarrow k_2 \Rightarrow \ldots \Rightarrow k_n \Rightarrow *$ geschrieben werden kann, kann das für diese Darstellung auch äquivalent als $\mathrm{o}(*) = 0$, $\mathrm{o}(k_1 \Rightarrow k_2 \Rightarrow \ldots \Rightarrow k_n \Rightarrow *) = \max_{i\in\{1,\ldots,n\}} \mathrm{o}(k_i) + 1$ geschrieben werden.

Der Rang eines Types ist der höchste Rang eines in ihm vorkommenden Subtypes (nur Lambdaabstraktionen können höheren Rang haben als die in ihnen vorkommenden Konstruktoren).

Der Rang eines Terms ist der höchste Rang eines in ihm vorkommenden Types.

Wie [12] schreibe ich $\mathrm{F}_\omega^{(n)}$ für ein auf Terme vom Rang $\leq n$ beschränktes System F_ω.

(Hinweis: In einigen Artikeln (z.B. [10, 4, 5, 8]) werden die Begriffe anders definiert, der Rang von $\mathrm{o}(*) = 1$ etc.)

Der Rang von $((* \Rightarrow *) \Rightarrow *) \Rightarrow *$ist also 3. $\lambda A^{*\Rightarrow *}.T^*$ hat den Rang 2.

1.2 System F_ω mit CASE und PAIR[1]

1.2.1 Curry Style

$$\frac{}{\Gamma \vdash x : \Gamma(x)}(\text{VAR})$$

$$\frac{\Gamma \vdash r : \rho \to \sigma \quad \Gamma \vdash s : \rho}{\Gamma \vdash rs : \sigma}(\text{APP})$$

$$\frac{\Gamma, x : \rho \vdash t : \sigma}{\Gamma \vdash \lambda x t : \rho \to \sigma}(\text{ABS})$$

$$\frac{\Gamma \vdash r : \sigma \quad \Gamma \vdash s : \rho}{\Gamma \vdash \langle r, s \rangle : \sigma \times \rho}(\text{PAIR})$$

$$\frac{\Gamma \vdash t : \sigma \times \rho}{\Gamma \vdash t\mathrm{L} : \sigma}(\text{PL})$$

$$\frac{\Gamma \vdash t : \sigma \times \rho}{\Gamma \vdash t\mathrm{R} : \rho}(\text{PR})$$

$$\frac{\Gamma \vdash t : \sigma + \rho \quad \Gamma, x : \sigma \vdash r : \gamma \quad \Gamma, y : \rho \vdash s : \gamma}{\Gamma \vdash t(x.r, y.s) : \gamma}(\text{CASE})$$

$$\frac{\Gamma \vdash t : \rho}{\Gamma \vdash \mathrm{INR} t : \sigma + \rho}(\text{INR})$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \mathrm{INL} t : \sigma + \rho}(\text{INL})$$

$$\frac{\Gamma \vdash t : \forall X \sigma}{\Gamma \vdash t : \sigma[X := \rho]}(\text{INST})$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash t : \forall X \sigma}\text{if X not free in } \Gamma(\text{GEN})$$

Das Kalkül ist also ein um die Regeln PAIR, PL,PR, CASE, INL, INR erweitertes System F_ω, in dem sich Konjunktionen und Disjunktionen durch diese syntaktischen Konstrukte direkt darstellen lassen.

1.2.2 syntaxgeleitete Version

Sie entsteht wieder durch Entfernen der Regeln INST und GEN und Erweitern der übrigen Regeln, so dass diese auch Ableitungen mit Typen, die zu den ursprünglichen in der Matching-Relation stehen, umfassen.

$$\frac{\Gamma(x) \leq \alpha}{\Gamma \vdash x : \alpha}(\text{VAR}')$$

$$\frac{\Gamma \vdash r : \rho \to \sigma \quad \Gamma \vdash s : \rho \quad \sigma \leq \alpha}{\Gamma \vdash rs : \forall \overrightarrow{X}.\alpha} \overrightarrow{X} \text{ not free in } \Gamma(\text{APP}')$$

[1] nach Ralph Matthes' [7]

$$\frac{\Gamma, x : \rho \vdash t : \sigma}{\Gamma \vdash \lambda x t : \forall \overrightarrow{X}.\rho \to \sigma} \overrightarrow{X} \text{ not free in } \Gamma(\text{ABS}')$$

$$\frac{\Gamma \vdash r : \sigma \quad \Gamma \vdash s : \rho}{\Gamma \vdash \langle r, s \rangle : \forall \overrightarrow{X}.\sigma \times \rho} \overrightarrow{X} \text{ not free in } \Gamma(\text{PAIR}')$$

$$\frac{\Gamma \vdash t : \sigma \times \rho \quad \sigma \leq \alpha}{\Gamma \vdash t\text{L} : \forall \overrightarrow{X}.\alpha} \overrightarrow{X} \text{ not free in } \Gamma(\text{PL}')$$

$$\frac{\Gamma \vdash t : \sigma \times \rho \quad \rho \leq \alpha}{\Gamma \vdash t\text{R} : \forall \overrightarrow{X}.\alpha} \overrightarrow{X} \text{ not free in } \Gamma(\text{PR}')$$

$$\frac{\Gamma \vdash t : \sigma + \rho \quad \Gamma, x : \sigma' \vdash r : \gamma \quad \sigma \leq \sigma' \quad \gamma \leq \alpha \quad \Gamma, y : \rho' \vdash s : \delta \quad \rho \leq \rho' \quad \delta \leq \alpha}{\Gamma \vdash t(x.r, y.s) : \forall \overrightarrow{X}.\alpha} \overrightarrow{X} \text{ not free in } \Gamma(\text{CASE}')$$

$$\frac{\Gamma \vdash t : \rho}{\Gamma \vdash \text{INR}t : \forall \overrightarrow{X}.\sigma + \rho} \overrightarrow{X} \text{ not free in } \Gamma(\text{INR}')$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \text{INL}t : \forall \overrightarrow{X}.\sigma + \rho} \overrightarrow{X} \text{ not free in } \Gamma(\text{INL}')$$

1.2.3 Äquivalenzbeweis

Die Äquivalenz der syntaxgeleiteten Version mit der ursprünglichen Variante lässt sich wahrscheinlich analog wie für System F_ω beweisen. Zumindest die Korrektheit, dass also jede im syntaxgeleiteten Kalkül erfolgte Ableitung eine entsprechende im Original besitzt, ist offensichtlich, da man nur für jede Regel das entsprechende Original verwenden muss und die spezielleren, semiunifizierten Typen durch Anwendung von INST und GEN an Hand der Substitution einfach nachbauen.

1.2.4 Church Style

Für INST und GEN verwendet man wieder die Church Notation und erhält einen entscheidbaren Kalkül.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}(\text{VAR})$$

$$\frac{\Gamma \vdash r : \rho \to \sigma \quad \Gamma \vdash s : \rho}{\Gamma \vdash rs : \sigma}(\text{APP})$$

$$\frac{\Gamma, x : \rho \vdash t : \sigma}{\Gamma \vdash \lambda x t : \rho \to \sigma}(\text{ABS})$$

$$\frac{\Gamma \vdash r : \sigma \quad \Gamma \vdash s : \rho}{\Gamma \vdash \langle r, s \rangle : \sigma \times \rho}(\text{PAIR})$$

$$\frac{\Gamma \vdash t : \sigma \times \rho}{\Gamma \vdash t\text{L} : \sigma}(\text{PL})$$

$$\frac{\Gamma \vdash t : \sigma \times \rho}{\Gamma \vdash t\mathrm{R} : \rho}(\mathrm{PR})$$

$$\frac{\Gamma \vdash t : \sigma + \rho \quad \Gamma, x : \sigma \vdash r : \gamma \quad \Gamma, y : \rho \vdash s : \gamma}{\Gamma \vdash t(x.r, y.s) : \gamma}(\mathrm{CASE})$$

$$\frac{\Gamma \vdash t : \rho}{\Gamma \vdash \mathrm{INR}t : \sigma + \rho}(\mathrm{INR})$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \mathrm{INL}t : \sigma + \rho}(\mathrm{INL})$$

$$\frac{\Gamma \vdash t : \forall X\sigma}{\Gamma \vdash t\rho : \sigma[X := \rho]}(\mathrm{INST})$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \Lambda X t : \forall X\sigma}\text{if X not free in } \Gamma(\mathrm{GEN})$$

1.3 syntaxgeleitete Kalküle

Ein Kalkül ist syntaxgeleitet, wenn sich aus dem Aufbau eines Terms eindeutig die verwendete Regel zu seiner Herleitung im Kalkül ergibt. Das führt (beweisbar durch strukturelle Induktion) dazu, dass der Ableitungsbaum für seine Typkorrektheit strukturgleich dem Termbaum ist.

Für die syntaxgeleitete Variante von F_ω gilt nun auch wie für jeden syntaxgeleiteten Kalkül, dass sich aus dem Wurzelknoten des Termbaum zwingend die Herleitungsregel ergibt, was im originalen System F_ω nicht gilt, da man an jeder Stelle Typvariablengeneralisierungen und -instanziierungen verwenden kann.

Diese Eigenschaft ist natürlich für die Typprüfung und -gewinnung günstig, da man nur die korrekte Anwendung einer Regel prüfen muss. Um sie für das System F_ω, das sie wegen der Regeln INST und GEN, die nur den Typ und nicht den Term modifizieren, nicht besitzt, zu erhalten, entfernt man entweder diese Regeln (durch Integration in die restlichen, verwendet in [10], 1.1.4, 1.2.2) oder ersetzt sie durch termmodifizierende (wie im Church-Kalkül 1.1.3, 1.2.4).

Der Church-Kalkül ist prinzipiell syntaxgeleitet, bei dem der Kalkül für die korrekten Terme identisch ist mit dem Kalkül für die typkorrekten. Das wird erreicht durch die syntaktischen Konstrukte für Typvariablengeneralisierungen und -instanziierungen.

1.4 annotierte Syntaxbäume

Das übliche Verfahren zur Typgewinnung und -prüfung ist die Verwendung eines mit dem Typ des jeweiligen Subterms annotierten Termbaums, für den man bei einem syntaxgeleiteten Kalkül entweder zur Typprüfung die korrekte Anwendung der jeweiligen Ableitungsregel überprüft oder zur Typgewinnung leere Knoten mit neuen Variablen beschriftet und die aus dem Typisierungskalkül sich ergebenden Constraints zu lösen versucht.

1.5 Algorithmen und Semi-Algorithmen zur Typprüfung

Aus den oben dargestellten syntaxgeleiteten Kalkülen (also entweder Church-Kalküle oder die syntaxgeleiteten Varianten der Curry-Kalküle) ergeben sich kanonisch Algorithmen zur Typprüfung von Termen, die diesen Typsystemen entsprechen sollen. Man betrachtet die Wurzel eines Termbaums, erhält daraus die einzig mögliche Herleitungsregel und prüft deren korrekte Anwendung, was für Church-Kalküle der Überprüfung der β-Gleichheit unter einer bekannten Substitution, korrekter Generalisierung, β-Gleichheit von Teiltypen (was alles auch praktisch entscheidbar ist) entspricht, für Curry-Kalküle dem Matchingproblem (der für unterschiedliche Ordnungen von Konstruktoren unentscheidbar oder entscheidbar) ist.

1.6 Der Semiunifikations-Algorithmus

1.6.1 Matching, Semiunifikation, Unifikation

Gegeben sei eine Menge von Gleichungen $E = \{M_{i\mathrm{l}} = M_{i\mathrm{r}} | i \in J\}$ und Mengen von Ungleichungen $I_n = \{N_{i\mathrm{l}} \leq N_{i\mathrm{r}} | i \in K_n\}$. Eine Substitution σ, für die es Quotientsubstitutionen ρ_n gibt mit den Eigenschaften $\forall i \in J.\sigma(M_{i\mathrm{l}}) = \sigma(M_{i\mathrm{r}})$ und $\forall n.\forall i \in K_n.\rho_n(\sigma(N_{i\mathrm{l}})) = \sigma(N_{i\mathrm{r}})$ ist ein (nicht-uniformer) Semiunifikator. Sind die ρ_n alle gleich, ist σ ein uniformer Semiunifikator, falls sie alle gleich der Identität sind, ist σ ein Unifikator. Ist σ die Identität, dann sind die ρ_n Matcher.

Ein Problemstellung kann immer durch eine einzige Gleichung ausgedrückt werden, so dass die Menge der Unifikatoren des ursprünglichen Problems identisch ist mit der der Gleichung. Ähnlich für uniforme Semiunifikatoren, bei denen immer Probleme auf eine Gleichung und eine Ungleichung reduziert werden können ([3], S.18).

1.6.2 Der Unifikations-Algorithmus von Huet

Huet hat in [5] einen Algorithmus zur Semiunifikation von einfach getypten Lambda-Termen vorgestellt.

Angepasst an die Verwendung von Allquantifikation im Term und den speziellen (binären und einzigen) Konstanten $\rightarrow, \times, +$wird dieser Algorithmus hier zur Semiunifikation von Typen verwendet. Dazu entfernt man die äusseren Allquantoren, „befreit" also deren gebundene Variablen und behandelt innere Allquantoren genauso wie Funktionssymbole, nur dass sie α-konvertierbar sind ($\forall X.\sigma$ durch $\forall Foo \lambda X.\sigma$ ersetzen oder direkt behandeln).

Man reduziert wie oben angesprochen ein Seminunifikationsproblem auf eine zu unifizierende Gleichung und löst diese dann mit diesem Algorithmus.

1.6.3 Matching-Algorithmus

Der in [4] - auf Baumautomaten beruhende - Algorithmus für das Matchingproblem bis zur Ordnung 3 entscheidet das Problem, das hier bei der Verifikation von voll anno-

tierten Termbäumen auftaucht, wenn über die korrekte Anwendung einer Regel eines syntaxgeleiteten Curry-Kalküls entschieden werden soll.

1.7 Ein Algorithmus zur Typgewinnung

Der Algorithmus geht in folgenden Schritten vor: Die Teile des Termbaums ohne Typinformation werden mit frischen Typvariablen annotiert. Dann stellt man mit Hilfe des syntaxgeleiteten Kalküls eine Menge von Gleichungen und Ungleichungen auf, die man mit dem Huet-Algorithmus löst. Der daraus entstehende Termbaum lässt sich für Ordnungen ≤ 3 unter zuhilfenahme des Matching-Algorthmus als korrekt annotiert verifizieren.

Bei dieser „naiven" Vorgehensweise tritt allerdings auch das in [3], S.33 beschriebene Problem auf, zu dessen Behebung die dort verwendete Methode der Variablenfixierung eingesetzt wird.

1.8 Hierarchie der betrachteten Probleme

Die Typprüfung in Church-Termen ist entscheidbar und algorithmisch einfach. Die Typprüfung von voll annotierten Curry-Termen ist bis zu einer gewissen Ordnung entscheidbar, allerdings sind die notwendigen Matching-Algorithmen nicht trivial. Typgewinnung für System F_ω ist unentscheidbar, aber reduzierbar auf Unifikation und damit den relativ luziden Algorithmus von Huet.

1.9 Typsysteme von Programmiersprachen

1.9.1 bidirektionale Typprüfung

Der bidirektionale Typprüfungsalgorithmus in [11] ist eine modifizierte und für intersection types spezialisierte Variante, die für F_ω über das generelle Prinzip der Verschränkung von Typgewinnung und Typprüfung hinaus keine für mich erkennbare Einsichten bietet. Wie oben bereits dargestellt, ist auch die Korrektheit der Typisierung eines vollständig annotierten Termbaumes unentscheidbar, weswegen eine partielle Verwendung von Instantiierungen a la Church vermutlich das bessere Verfahren darstellt.

1.9.2 entscheidbare Typsysteme

Typsysteme, für die Implementierungen zur Typprüfung von funktionalen Programmiersprachen wie Standard ML oder Haskell existieren oder die zu diesem Zweck entworfen wurden, stellt Henglein in seiner Doktorarbeit [3] vor. Dazu zählen der Curry-Hindley, Damas-Milner, Milner-Mycroft und der Flat Milner-Mycroft Kalkül.

Relevant sind diese Systeme und Implementierungen für meine Fragestellung nicht, da die Zielsetzungen zu unterschiedlich sind, d.h. wegen der Notwendigkeit von terminierenden Algorithmen werden nur viel schwächere Typsysteme als F_ω untersucht.

1.9.3 Ponder

Es fehlt leider auch „Ponder"[2], vermutlich die erste (und einzige?) Sprache mit einem Typsystem so mächtig wie F_ω, über die ich nichts weiter recherchieren konnte.

[2] Stichwort „Ponder" im FOLDOC [2]

2 Die Implementierung

erfolgte in Haskell und ist beschränkt auf einen Typchecker für den Church-Kalkül für F_ω mit CASE und PAIR. Im übrigen kann ich mich auf den Artikel [6] berufen, der behauptet auf die „esoterischeren" Sprachkonstrukte in Haskell zu verzichten und nur Monads in allen Varianten zu verwenden.

2.1 Datentypen und Abstraktion

Verwendet werden ein Datentyp Exp für partiell annotierte Termbäume, Type für partiell annotierte Typbäume, Kind für Kindbäume.

Die offensichtliche Abstraktion der Verwendung einer gemeinsamen Struktur für die (durch Typen) getypten Terme und (durch Kinds) getypten Typen findet in der class Lambda ihren Ausdruck. Dabei zeigen sich allerdings eine Reihe von Problemen, da Haskell weder classes mit mehreren Parametern zulässt, noch Instanzdeklarationen mit Typsynonymen oder Typapplikationen.

Durch die überladenen Funktionen destr und constr werden die Baumdatentypen in durch Listen immitierte B äume verwandelt und umgekehrt, wobei noch eventuelle Variablenbindungen vermerkt werden.

2.2 Der Parser

ist mit dem Parsergenerator happy für Haskell erstellt und benutzt einen handgeschriebenen Lexer. Die Grammatik orientiert sich grundsätzlich an der üblichen Notation für getypte Lambdaterme mit „:" als dem Junktor zwischen Termen und Typen, „\" als λ-Symbol wie in Haskell, und dem Kaufmannsund „&" als $\forall$. Variablen dürfen länger als ein Zeichen sein und bestehen aus Buchstaben, nach dem ersten Zeichen auch aus Ziffern. Sie werden durch Zeichen, die keine Buchstaben oder Ziffern sind, voneinander getrennt. Großgeschriebene Variablen sind Termvariablen, kleingeschriebene Typvariablen.

Ein Unterschied zur gewöhnlichen Notation ist, dass der Punkt „." ein syntaktisches Element ist, das in jeder Variablenbindung vorkommt. Er hat den üblichen Effekt, die Bindung so weit nach rechts wie möglich auszudehnen.

Der Parser soll zusammen mit der show Funktion eine Isomorphie zwischen den als Daten und den als Strings repäsentierten Lambdatermen erzeugen, wobei überflüssige Klammern und whitespace ausser Betracht bleiben.

2.3 Der Pretty-Printer

kommt in mehreren Varianten vor. Zum einen als show-Funktion, die die abstrakten Syntaxbäume wieder in (parsebare) Strings übersetzt (und damit „die" Repräsentation eines Terms ist). Daneben gibt es noch explicit, die den Term vollständig klammert und alle Klammerersparnisregeln ignoriert. latex übersetzt die Terme in LaTeX-Ausdrücke.

2.4 Hilfsfunktionen

β-Reduktion, freie Variablen, gebundene Variablen, α-Konversion,

2.5 Der Algorithmus zur vollständigen Annotation

Die Funktion annotateFull ergänzt die Typen eines jeden Teilterms, der noch keine Typisierung besitzt und überprüft die bereits annotierten.

2.6 Der Typprüfer

Die Funktion checkChurch prüft einen voll annotierten Baum auf Korrektheit der Annotation. Dabei werden zuerst rekursiv die Teilterme geprüft und dann an Hand der Regel, die dem Operator der Wurzel zugeordnet ist, eben diese. Aus den Regeln (siehe 1.2.4) ergeben sich dann die Programmkonstrukte

2.7 Beispiele

Das ganze System ist getestet sowohl mit `The Glorious Glasgow Haskell Compilation System, version 5.04.2`

als auch mit `Hugs Version November 2002` . Da keine systemspezifische Erweiterungen verwendet wurden, sollte es mit allem arbeiten, was Haskell 98 versteht. Die Beispiele sind theoretisch banal und dienen zum Verständnis des Systems und der Fehlerbeseitigung.

Die Tabellen stellen eine Auflistung aller Teilterme eines zu prüfenden Terms dar, wobei dann kontrolliert wird, ob der annotierte Typ sich korrekt aus denen der inneren Terme ergibt (Das ist mit der Gleichheitsrelation jetzt auch nur interessant, um Fehler im Programm zu finden, bei komplizierteren Relationen gilt das nicht mehr).

2.7.1 Kombinatoren

Die Beispiele bestehen aus den Kombinatoren S ($=\lambda x.\lambda y.\lambda z.xz(yz)$) und K ($=\lambda x.\lambda y.x$) aus dem Kombinatorenkalkül,

```
\x.\y.\z.x z (y z)
```

$\lambda x.\lambda y.\lambda z.xz(yz)$

```
\x.\y.x
```

$\lambda x.\lambda y.x$

```
(\x.\y.\z.x z (y z)) (\x.\y.x) \x.\y.x
```

$(\lambda x.\lambda y.\lambda z.xz(yz))(\lambda x.\lambda y.x)\lambda x.\lambda y.x$

Die Betareduktion funktioniert zumindest bei der Identität I$=_\beta$SKK

```
(\x.\y.\z.x z (y z)) (\x.\y.x) \x.\y.x
```

$(\lambda x.\lambda y.\lambda z.xz(yz))(\lambda x.\lambda y.x)\lambda x.\lambda y.x$

$\rightarrow_\beta$

```
(\y.\z.(\x.\y.x) z (y z)) \x.\y.x
```

$(\lambda y.\lambda z.(\lambda x.\lambda y.x)z(yz))\lambda x.\lambda y.x$

$\rightarrow_\beta$

```
\z.(\x.\y.x) z ((\x.\y.x) z)
```

$\lambda z.(\lambda x.\lambda y.x)z((\lambda x.\lambda y.x)z)$

$\rightarrow_\beta$

```
\z.(\y.z) ((\x.\y.x) z)
```

$\lambda z.(\lambda y.z)((\lambda x.\lambda y.x)z)$

$\rightarrow_\beta$

```
\z.z
```

$\lambda z.z$

2.7.2 Church-Ziffern

ein paar Church-Ziffern (die Church-Ziffer n hat die Form: $\lambda x.\lambda f.f^n x = \lambda x.\lambda f.\underbrace{f \ldots f}\, x$) als grosse Terme

```
\f.\x.x
```

$\lambda f.\lambda x.x$

```
\f.\x.f x
```

$\lambda f.\lambda x.fx$

```
\f.\x.f (f (f x))
```

$\lambda f.\lambda x.f(f(fx))$

```
\f.\x.f (f (f (f (f (f x))))))
```

$\lambda f.\lambda x.f(f(f(f(f(fx))))))$

```
\f:(A^*->A^*)^*.\x:A^*.((f:(A^*->A^*)^*) ((f:(A^*->A^*)^*) ((f:(A^*->A^*)^*) (x:
A^*):A^*):A^*):A^*):(((A^*->A^*)^*)->(A^*->A^*)^*)^*
```

$\lambda f : (A^* \to A^*)^*.\lambda x : A^*.((f : (A^* \to A^*)^*)((f : (A^* \to A^*)^*)((f : (A^* \to A^*)^*)(x : A^*) : A^*) : A^*) : A^*) : (((A^* \to A^*)^*) \to (A^* \to A^*)^*)^*$

```
\f:(A^*->A^*)^*.\x:A^*.((f:(A^*->A^*)^*) ((f:(A^*->A^*)^*) ((f:(A^*->A^*)^*) ((f
:(A^*->A^*)^*) ((f:(A^*->A^*)^*) (x:A^*):A^*):A^*):A^*):A^*):A^*):(((A^*->A^*)^*
)->(A^*->A^*)^*)^*
```

$\lambda f : (A^* \to A^*)^*.\lambda x : A^*.((f : (A^* \to A^*)^*)((f : (A^* \to A^*)^*)((f : (A^* \to A^*)^*)((f : (A^* \to A^*)^*)((f : (A^* \to A^*)^*)(x : A^*) : A^*) : A^*) : A^*) : A^*) : A^*) : (((A^* \to A^*)^*) \to (A^* \to A^*)^*)^*$

und aus den von Urzyczyn verwendeten Zweierketten ($22 \ldots 2\mathrm{K}$) der Länge n bei denen ein Ausdruck aus n-mal der Church-Ziffer 2 auf K angewendet, was an die Ackermann-Funktion erinnert (urzy $n = (\text{church } 2)^n$ K).

```
(\f.\x.f (f x)) \x.\y.x
```

$(\lambda f.\lambda x.f(fx))\lambda x.\lambda y.x$

```
(\f.\x.f (f x)) (\f.\x.f (f x)) \x.\y.x
```

$(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(fx))\lambda x.\lambda y.x$

```
(\f.\x.f (f x)) (\f.\x.f (f x)) (\f.\x.f (f x)) (\f.\x.f (f x)) (\f.\x.f (f x))
(\f.\x.f (f x)) \x.\y.x
```

$(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(fx))\lambda x.\lambda y.x$

2.7.3 Verschiedenes

Zurückweisung einer Übergeneralisierung

```
/\A^*.(x:A^*)
```

$\Lambda A^*.(x : A^*)$

```
/\A^*.(x:A^*):(&A^*.A^*)^* :
ERROR: Illegal Generalisation. TVar:A^* free in x:A^*.
freevar e: [x:A^*]
freetv (freevar e): [A^*]
---
```

Falsches als Falsches erkannt

Getypte Kombinatoren S K

Term		
computed Type	annotated Type	Relation: ==
$(\Lambda Z^*.\Lambda Y^*.\Lambda X^*.\lambda x : Z^* \to Y^* \to X^*.\lambda y : Z^* \to Y^*.\lambda z : Z^*.xz(yz))Xk^*Yk^*Xk^*\lambda x : Xk^*.\lambda y : Yk^*.x$		
$((((\Lambda Z^*.(\Lambda Y^*.(\Lambda X^*.(\lambda x : (Z^* \to (Y^* \to X^*)^*)^*.(\lambda y : (Z^* \to Y^*)^*.(\lambda z : Z^*.(((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*) : (Z^* \to X^*)^*) : (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*) : (\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*) : (\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*) : (\forall Z^*.(\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*)Xk^* : (\forall Y^*.(\forall X^*.(((Xk^* \to (Y^* \to X^*)^*)^*) \to (((Xk^* \to Y^*)^*) \to (Xk^* \to X^*)^*)^*)^*)^*)Yk^* : (\forall X^*.(((Xk^* \to (Yk^* \to X^*)^*)^*) \to (((Xk^* \to Yk^*)^*) \to (Xk^* \to X^*)^*)^*)^*)Xk^* : (((Xk^* \to (Yk^* \to Xk^*)^*)^*) \to (((Xk^* \to Yk^*)^*) \to (Xk^* \to Xk^*)^*)^*)(\lambda x : Xk^*.(\lambda y : Yk^*.(x : Xk^*) : (Yk^* \to Xk^*)^*) : (Xk^* \to (Yk^* \to Xk^*)^*)^*) : (((Xk^* \to Yk^*)^*) \to (Xk^* \to Xk^*)^*)^*$		
$(((Xk^* \to Yk^*)^*) \to (Xk^* \to Xk^*)^*)^*$	$(((Xk^* \to Yk^*)^*) \to (Xk^* \to Xk^*)^*)^*$	True
$(((\Lambda Z^*.(\Lambda Y^*.(\Lambda X^*.(\lambda x : (Z^* \to (Y^* \to X^*)^*)^*.(\lambda y : (Z^* \to Y^*)^*.(\lambda z : Z^*.(((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*) : (Z^* \to X^*)^*) : (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*) : (\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*) : (\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*) : (\forall Z^*.(\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*)Xk^* : (\forall Y^*.(\forall X^*.(((Xk^* \to (Y^* \to X^*)^*)^*) \to (((Xk^* \to Y^*)^*) \to (Xk^* \to X^*)^*)^*)^*)^*)Yk^* : (\forall X^*.(((Xk^* \to (Yk^* \to X^*)^*)^*) \to (((Xk^* \to Yk^*)^*) \to (Xk^* \to X^*)^*)^*)^*)Xk^* : (((Xk^* \to (Yk^* \to Xk^*)^*)^*) \to (((Xk^* \to Yk^*)^*) \to (Xk^* \to Xk^*)^*)^*)^*$		
$(((Xk^* \to (Yk^* \to Xk^*)^*)^*) \to (((Xk^* \to Yk^*)^*) \to (Xk^* \to Xk^*)^*)^*)^*$	$(((Xk^* \to (Yk^* \to Xk^*)^*)^*) \to (((Xk^* \to Yk^*)^*) \to (Xk^* \to Xk^*)^*)^*)^*$	True
$((\Lambda Z^*.(\Lambda Y^*.(\Lambda X^*.(\lambda x : (Z^* \to (Y^* \to X^*)^*)^*.(\lambda y : (Z^* \to Y^*)^*.(\lambda z : Z^*.(((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*) : (Z^* \to X^*)^*) : (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*) : (\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*) : (\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*) : (\forall Z^*.(\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*)Xk^* : (\forall Y^*.(\forall X^*.(((Xk^* \to (Y^* \to X^*)^*)^*) \to (((Xk^* \to Y^*)^*) \to (Xk^* \to X^*)^*)^*)^*)^*)Yk^* : (\forall X^*.(((Xk^* \to (Yk^* \to X^*)^*)^*) \to (((Xk^* \to Yk^*)^*) \to (Xk^* \to X^*)^*)^*)^*$		

$(\forall X^*.(((Xk^* \to (Yk^* \to X^*)^*)^*) \to (((Xk^* \to Yk^*)^*) \to (Xk^* \to X^*)^*)^*)^*$	$(\forall X^*.(((Xk^* \to (Yk^* \to X^*)^*)^*) \to (((Xk^* \to Yk^*)^*) \to (Xk^* \to X^*)^*)^*)^*$	True
$(\Lambda Z^*.(\Lambda Y^*.(\Lambda X^*.(\lambda x : (Z^* \to (Y^* \to X^*)^*)^*.(\lambda y : (Z^* \to Y^*)^*.(\lambda z : Z^*.(((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*) : (Z^* \to X^*)^*) : (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*) : (\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*) : (\forall Z^*.(\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*)^*)Xk^* : (\forall Y^*.(\forall X^*.(((Xk^* \to (Y^* \to X^*)^*)^*) \to (((Xk^* \to Y^*)^*) \to (Xk^* \to X^*)^*)^*)^*)^*$		
$(\forall Y^*.(\forall X^*.(((Xk^* \to (Y^* \to X^*)^*)^*) \to (((Xk^* \to Y^*)^*) \to (Xk^* \to X^*)^*)^*)^*)^*$	$(\forall Y^*.(\forall X^*.(((Xk^* \to (Y^* \to X^*)^*)^*) \to (((Xk^* \to Y^*)^*) \to (Xk^* \to X^*)^*)^*)^*)^*$	True
$\Lambda Z^*.(\Lambda Y^*.(\Lambda X^*.(\lambda x : (Z^* \to (Y^* \to X^*)^*)^*.(\lambda y : (Z^* \to Y^*)^*.(\lambda z : Z^*.(((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*) : (Z^* \to X^*)^*) : (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*) : (\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*) : (\forall Z^*.(\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*)^*$		
$(\forall Z^*.(\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*)^*$	$(\forall Z^*.(\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*)^*$	True
$\Lambda Y^*.(\Lambda X^*.(\lambda x : (Z^* \to (Y^* \to X^*)^*)^*.(\lambda y : (Z^* \to Y^*)^*.(\lambda z : Z^*.(((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*) : (Z^* \to X^*)^*) : (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*) : (\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*$		
$(\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*$	$(\forall Y^*.(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*)^*$	True
$\Lambda X^*.(\lambda x : (Z^* \to (Y^* \to X^*)^*)^*.(\lambda y : (Z^* \to Y^*)^*.(\lambda z : Z^*.(((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*) : (Z^* \to X^*)^*) : (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*$		
$(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*$	$(\forall X^*.(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*$	True

$\lambda x : (Z^* \to (Y^* \to X^*)^*)^*.(\lambda y : (Z^* \to Y^*)^*.(\lambda z : Z^*.(((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*) : (Z^* \to X^*)^*) : (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*) : (((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*$		
$(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*$	$(((Z^* \to (Y^* \to X^*)^*)^*) \to (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*)^*$	True
$\lambda y : (Z^* \to Y^*)^*.(\lambda z : Z^*.(((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*) : (Z^* \to X^*)^*) : (((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*$		
$(((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*$	$(((Z^* \to Y^*)^*) \to (Z^* \to X^*)^*)^*$	True
$\lambda z : Z^*.(((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*) : (Z^* \to X^*)^*$		
$(Z^* \to X^*)^*$	$(Z^* \to X^*)^*$	True
$((x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*)((y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*) : X^*$		
X^*	X^*	True
$(x : (Z^* \to (Y^* \to X^*)^*)^*)(z : Z^*) : (Y^* \to X^*)^*$		
$(Y^* \to X^*)^*$	$(Y^* \to X^*)^*$	True
$x : (Z^* \to (Y^* \to X^*)^*)^*$		
$(Z^* \to (Y^* \to X^*)^*)^*$	$(Z^* \to (Y^* \to X^*)^*)^*$	True
$z : Z^*$		
Z^*	Z^*	True
$(y : (Z^* \to Y^*)^*)(z : Z^*) : Y^*$		
Y^*	Y^*	True
$y : (Z^* \to Y^*)^*$		
$(Z^* \to Y^*)^*$	$(Z^* \to Y^*)^*$	True
$z : Z^*$		
Z^*	Z^*	True
$\lambda x : Xk^*.(\lambda y : Yk^*.(x : Xk^*) : (Yk^* \to Xk^*)^*) : (Xk^* \to (Yk^* \to Xk^*)^*)^*$		
$(Xk^* \to (Yk^* \to Xk^*)^*)^*$	$(Xk^* \to (Yk^* \to Xk^*)^*)^*$	True
$\lambda y : Yk^*.(x : Xk^*) : (Yk^* \to Xk^*)^*$		
$(Yk^* \to Xk^*)^*$	$(Yk^* \to Xk^*)^*$	True
$x : Xk^*$		
Xk^*	Xk^*	True

Ein halber SKK ... als Beispiel für einen großen Term mit Typen

Typ von Urzyczyns 22K

Term		
computed Type	annotated Type	Relation: ==
$\lambda f : (\forall B^{*\Rightarrow *}.(\forall A^*.A^* \to BA^*) \to \forall A^*.A^* \to B(BA^*)).\lambda x : (\forall A^*.A^* \to B^{*\Rightarrow *}A^*).fB^{*\Rightarrow *}x$		

$\lambda f : (\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*).(\lambda x : (\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*).(((f : (\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*)B^{*\Rightarrow *} : (((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)(x : (\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) : (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*) : (((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*) : (((\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*) \to (((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$		
$(((\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*) \to (((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$	$(((\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*) \to (((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$	True
$\lambda x : (\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*).(((f : (\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*)B^{*\Rightarrow *} : (((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)(x : (\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) : (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*) : (((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$		
$(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$	$(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$	True
$((f : (\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*)B^{*\Rightarrow *} : (((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)(x : (\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) : (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*$		
$(\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*$	$(\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*$	True
$(f : (\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*)B^{*\Rightarrow *} : (((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*$		
$(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$	$(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$	True
$f : (\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$		
$(\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$	$(\forall B^{*\Rightarrow *}.(((\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*) \to (\forall A^*.(A^* \to (B^{*\Rightarrow *}((B^{*\Rightarrow *}A^*)^*))^*)^*)^*)^*$	True
$x : (\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*$		
$(\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*$	$(\forall A^*.(A^* \to (B^{*\Rightarrow *}A^*)^*)^*)^*$	True

Beweis der Typisierbarkeit von 22K, eine etwas anspruchsvollere Ableitung

Einfache Paarbildung

Term		
computed Type	annotated Type	Relation: ==
$\lambda x : A^*. < x, x >$		
$\lambda x : A^*.(< x : A^*, x : A^* >: (A^* \times A^*)^*) : (A^* \to (A^* \times A^*)^*)^*$		
$(A^* \to (A^* \times A^*)^*)^*$	$(A^* \to (A^* \times A^*)^*)^*$	True
$< x : A^*, x : A^* >: (A^* \times A^*)^*$		
$(A^* \times A^*)^*$	$(A^* \times A^*)^*$	True
$x : A^*$		
A^*	A^*	True
$x : A^*$		
A^*	A^*	True

Beweist nichts ...

2.7.4 Montoniezeugen

Auch einfache Exemplare von Matthes' Monotoniezeugen (aus [9]) lassen sich verifizieren

Term		
computed Type	annotated Type	Relation: ==
$\Lambda Am^*.\Lambda A^*.\lambda f : (Am^* \to A^*)^*.\lambda x : Am^*.fx : (\lambda F^{*\Rightarrow *}.\forall Am^*.\forall A^*.(Am \to A) \to FAm \to FA)\lambda A^*.A$		
$\Lambda Am^*.(\Lambda A^*.(\lambda f : (Am^* \to A^*)^*.(\lambda x : Am^*.((f : (Am^* \to A^*)^*)(x : Am^*) : A^*) : (Am^* \to A^*)^*) : (((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*) : (\forall A^*.(((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*)^*) : (\forall Am^*.(\forall A^*.(((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*)^*$		
$(\forall Am^*.(\forall A^*.(((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*)^*$	$(\forall Am^*.(\forall A^*.(((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*)^*$	True
$\Lambda A^*.(\lambda f : (Am^* \to A^*)^*.(\lambda x : Am^*.((f : (Am^* \to A^*)^*)(x : Am^*) : A^*) : (Am^* \to A^*)^*) : (((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*) : (\forall A^*.(((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*)^*$		
$(\forall A^*.(((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*)^*$	$(\forall A^*.(((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*)^*$	True
$\lambda f : (Am^* \to A^*)^*.(\lambda x : Am^*.((f : (Am^* \to A^*)^*)(x : Am^*) : A^*) : (Am^* \to A^*)^*) : (((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*$		
$(((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*$	$(((Am^* \to A^*)^*) \to (Am^* \to A^*)^*)^*$	True
$\lambda x : Am^*.((f : (Am^* \to A^*)^*)(x : Am^*) : A^*) : (Am^* \to A^*)^*$		
$(Am^* \to A^*)^*$	$(Am^* \to A^*)^*$	True
$(f : (Am^* \to A^*)^*)(x : Am^*) : A^*$		
A^*	A^*	True
$f : (Am^* \to A^*)^*$		
$(Am^* \to A^*)^*$	$(Am^* \to A^*)^*$	True
$x : Am^*$		
Am^*	Am^*	True

Die Identität ist isoton, Punkt 1.3 in [9]

Term		
computed Type	annotated Type	Relation: ==
$\Lambda A^*.\Lambda B^*.\lambda f : A^* \to B^*.\lambda p : A^* \times A^*. < f(pL), f(pR) >: (\lambda F^{*\Rightarrow *}.\forall Am^*.\forall A^*.(Am \to A) \to FAm \to FA)\lambda X^*.X \times X$		
$\Lambda A^*.(\Lambda B^*.(\lambda f : (A^* \to B^*)^*.(\lambda p : (A^* \times A^*)^*.(< (f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)L : A^*) : B^*, (f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)R : A^*) : B^* >: (B^* \times B^*)^*) : ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*) : (((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*)^*) : (\forall B^*.(((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*)^*) : (\forall A^*.(\forall B^*.(((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*)^*)^*$		
$(\forall A^*.(\forall B^*.(((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*)^*)^*$	$(\forall A^*.(\forall B^*.(((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*)^*)^*$	True
$\Lambda B^*.(\lambda f : (A^* \to B^*)^*.(\lambda p : (A^* \times A^*)^*.(< (f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)L : A^*) : B^*, (f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)R : A^*) : B^* >: (B^* \times B^*)^*) : ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*) : (((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*)^*) : (\forall B^*.(((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*)^*$		
$(\forall B^*.(((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*)^*$	$(\forall B^*.(((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*)^*$	True
$\lambda f : (A^* \to B^*)^*.(\lambda p : (A^* \times A^*)^*.(< (f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)L : A^*) : B^*, (f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)R : A^*) : B^* >: (B^* \times B^*)^*) : ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*) : (((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*$		
$(((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*$	$(((A^* \to B^*)^*) \to ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*$	True
$\lambda p : (A^* \times A^*)^*.(< (f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)L : A^*) : B^*, (f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)R : A^*) : B^* >: (B^* \times B^*)^*) : ((A^* \times A^*)^* \to (B^* \times B^*)^*)^*$		
$((A^* \times A^*)^* \to (B^* \times B^*)^*)^*$	$((A^* \times A^*)^* \to (B^* \times B^*)^*)^*$	True
$< (f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)L : A^*) : B^*, (f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)R : A^*) : B^* >: (B^* \times B^*)^*$		
$(B^* \times B^*)^*$	$(B^* \times B^*)^*$	True
$(f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)L : A^*) : B^*$		
B^*	B^*	True
$f : (A^* \to B^*)^*$		
$(A^* \to B^*)^*$	$(A^* \to B^*)^*$	True
$(p : (A^* \times A^*)^*)L : A^*$		
A^*	A^*	True
$p : (A^* \times A^*)^*$		
$(A^* \times A^*)^*$	$(A^* \times A^*)^*$	True
$(f : (A^* \to B^*)^*)((p : (A^* \times A^*)^*)R : A^*) : B^*$		
B^*	B^*	True
$f : (A^* \to B^*)^*$		
$(A^* \to B^*)^*$	$(A^* \to B^*)^*$	True
$(p : (A^* \times A^*)^*)R : A^*$		
A^*	A^*	True
$p : (A^* \times A^*)^*$		
$(A^* \times A^*)^*$	$(A^* \times A^*)^*$	True

Das Produkt ist isoton

Term		
computed Type	annotated Type	Relation: ==
$\Lambda A^*.\Lambda B^*.\lambda f : A^* \to B^*.\lambda c : A^* + A^*.(INL(c(cl : A^*.fcl : B^*, cr : A^*.fcr : B^*)) : B^* + B^*) : (\lambda F^{*\Rightarrow *}.\forall Am^*.\forall A^*.(Am \to A) \to FAm \to FA)\lambda X^*.X + X$		
$\Lambda A^*.(\Lambda B^*.(\lambda f : (A^* \to B^*)^*.(\lambda c : (A^* + A^*)^*.(INL((c : (A^* + A^*)^*)(cl : A^*.(f : (A^* \to B^*)^*)(cl : A^*) : B^*, cr : A^*.(f : (A^* \to B^*)^*)(cr : A^*) : B^*) : B^*) : (B^* + B^*)^*) : ((A^* + A^*)^* \to (B^* + B^*)^*)^*) : (((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*) : (\forall B^*.(((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*) : (\forall A^*.(\forall B^*.(((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*)^*$		
$(\forall A^*.(\forall B^*.(((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*)^*$	$(\forall A^*.(\forall B^*.(((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*)^*$	True
$\Lambda B^*.(\lambda f : (A^* \to B^*)^*.(\lambda c : (A^* + A^*)^*.(INL((c : (A^* + A^*)^*)(cl : A^*.(f : (A^* \to B^*)^*)(cl : A^*) : B^*, cr : A^*.(f : (A^* \to B^*)^*)(cr : A^*) : B^*) : B^*) : (B^* + B^*)^*) : ((A^* + A^*)^* \to (B^* + B^*)^*)^*) : (((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*) : (\forall B^*.(((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*$		
$(\forall B^*.(((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*$	$(\forall B^*.(((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*$	True
$\lambda f : (A^* \to B^*)^*.(\lambda c : (A^* + A^*)^*.(INL((c : (A^* + A^*)^*)(cl : A^*.(f : (A^* \to B^*)^*)(cl : A^*) : B^*, cr : A^*.(f : (A^* \to B^*)^*)(cr : A^*) : B^*) : B^*) : (B^* + B^*)^*) : ((A^* + A^*)^* \to (B^* + B^*)^*)^*) : (((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*$		
$(((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*$	$(((A^* \to B^*)^*) \to ((A^* + A^*)^* \to (B^* + B^*)^*)^*)^*$	True
$\lambda c : (A^* + A^*)^*.(INL((c : (A^* + A^*)^*)(cl : A^*.(f : (A^* \to B^*)^*)(cl : A^*) : B^*, cr : A^*.(f : (A^* \to B^*)^*)(cr : A^*) : B^*) : B^*) : (B^* + B^*)^*) : ((A^* + A^*)^* \to (B^* + B^*)^*)^*$		
$((A^* + A^*)^* \to (B^* + B^*)^*)^*$	$((A^* + A^*)^* \to (B^* + B^*)^*)^*$	True
$INL((c : (A^* + A^*)^*)(cl : A^*.(f : (A^* \to B^*)^*)(cl : A^*) : B^*, cr : A^*.(f : (A^* \to B^*)^*)(cr : A^*) : B^*) : B^*) : (B^* + B^*)^*$		
$(B^* + B^*)^*$	$(B^* + B^*)^*$	True
$(c : (A^* + A^*)^*)(cl : A^*.(f : (A^* \to B^*)^*)(cl : A^*) : B^*, cr : A^*.(f : (A^* \to B^*)^*)(cr : A^*) : B^*) : B^*$		
B^*	B^*	True
$c : (A^* + A^*)^*$		
$(A^* + A^*)^*$	$(A^* + A^*)^*$	True
$(f : (A^* \to B^*)^*)(cl : A^*) : B^*$		
B^*	B^*	True
$f : (A^* \to B^*)^*$		
$(A^* \to B^*)^*$	$(A^* \to B^*)^*$	True
$cl : A^*$		
A^*	A^*	True
$(f : (A^* \to B^*)^*)(cr : A^*) : B^*$		
B^*	B^*	True
$f : (A^* \to B^*)^*$		
$(A^* \to B^*)^*$	$(A^* \to B^*)^*$	True
$cr : A^*$		
A^*	A^*	True

Die Summe ist isoton

Term		
computed Type	annotated Type	Relation: ==
$\Lambda B^*.\Lambda Am^*.\Lambda A^*.\lambda f : Am^* \to A^*.\lambda x : B^*.x$		
$\Lambda B^*.(\Lambda Am^*.(\Lambda A^*.(\lambda f : (Am^* \to A^*)^*.(\lambda x : B^*.(x : B^*) : (B^* \to B^*)^*) : (((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*) : (\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*)^*) : (\forall Am^*.(\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*)^*) : (\forall B^*.(\forall Am^*.(\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*)^*)^*$		
$(\forall B^*.(\forall Am^*.(\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*)^*)^*$	$(\forall B^*.(\forall Am^*.(\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*)^*)^*$	True
$\Lambda Am^*.(\Lambda A^*.(\lambda f : (Am^* \to A^*)^*.(\lambda x : B^*.(x : B^*) : (B^* \to B^*)^*) : (((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*) : (\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*) : (\forall Am^*.(\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*)^*$		
$(\forall Am^*.(\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*)^*$	$(\forall Am^*.(\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*)^*$	True
$\Lambda A^*.(\lambda f : (Am^* \to A^*)^*.(\lambda x : B^*.(x : B^*) : (B^* \to B^*)^*) : (((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*) : (\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*$		
$(\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*$	$(\forall A^*.(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*$	True
$\lambda f : (Am^* \to A^*)^*.(\lambda x : B^*.(x : B^*) : (B^* \to B^*)^*) : (((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*$		
$(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*$	$(((Am^* \to A^*)^*) \to (B^* \to B^*)^*)^*$	True
$\lambda x : B^*.(x : B^*) : (B^* \to B^*)^*$		
$(B^* \to B^*)^*$	$(B^* \to B^*)^*$	True
$x : B^*$		
B^*	B^*	True

Konstanten sind isoton, Punkt 1.2 in [9]

3 Anhang

3.1 Die Programme

3.1.1 Die Datentypen mit show

```
-- these are datatypes for system F_omega typed lambda expressions with CASE
-- and PAIR rules. All a la Church with Maybe type assignments
-- they can also be used to represent Curry terms
-- $Id: CPFwDatatypes.hs,v 1.12 2002/05/01 18:29:35 markus Exp $

module CPFwDatatypes where

import List(notElem, intersect)
import Maybe(fromJust, isJust)

infix :::
infix ::^

-- Exp, Type and Kind extraction
-- smells like a class, but Exp has to be a "real" type, so i don't know how
-- to create one

tm :: Exp->UntypedExp
tm (e ::: _) = e

ty :: Exp->Maybe Type
ty (_ ::: t) = t
jty = fromJust . ty

con :: Type->UnkindedType
con (t ::^ _) = t

kd :: Type->Maybe Kind
kd (_ ::^ k) = k
jkd = fromJust . kd

-- conditional show
cs b br = if b then br else ""
```

```
data OpStyle = Base | LeftAss | RightAss | Non | Post | Pre deriving Eq

data Binary = Function | Prod | Sum deriving Eq
instance Show Binary where
  show Function = "->"
  show Prod = "%"
  show Sum = "+"
instance Read Binary where
  readsPrec _ ('-':'>':r) = [(Function, r)]
  readsPrec _ ('%':r) = [(Prod, r)]
  readsPrec _ ('+':r) = [(Sum, r)]
  readsPrec _ _ = []

data LR = L | R deriving (Show, Read, Eq)

class Show_ a where
  show_ :: Bool -> a -> String

class Prec a where
  prec :: a -> (Int, OpStyle)

-- precProb(T) (not a class because of Haskell limitations)
-- if the precedence of the outer (upper in a parse tree) is higher or equal
-- as the inner and the operator style makes it possible, brackets are
-- neccesary to avoid "stealing"

data UntypedExp
      = Abs Exp Exp
      | Inst Exp Type
      | Gen Type Exp
      | App Exp Exp
      | Var String
      | Case Exp Exp Exp Exp Exp
      | InLR LR Exp
      | Pair Exp Exp
      | Lr LR Exp
     deriving Eq
instance Show UntypedExp where
  show = show_ True -- first arg says whether term is in front of a closing
                    -- bracket to bracket \ . correctly

instance Prec UntypedExp where
  prec (Abs _ _) = (1000, Pre)
```

```
  prec (Gen _ _) = (1000, Pre)
  prec (InLR _ _) = (500, Pre)
  prec (App _ _) = (100, LeftAss)
  prec (Inst _ _) = (100, Post)
  prec (Case _ _ _ _ _) = (700, Post)
  prec (Lr _ _) = (200, Post)
  prec (Var _) = (0, Base)
  prec (Pair _ _) = (0, Base)

precProb ops out (inn ::: t) =
     (fst (prec out) < fst (prec inn)
       && (snd (prec inn) `notElem` (ops `intersect` [Post, Pre])))
  || (fst (prec out) == fst (prec inn)
       && (snd (prec inn) `notElem` ops)
  || isJust t)

instance Show_ UntypedExp where
  -- dot structures (prefix), lowest prec so no testing neccesary
  show_ b (Abs v t) =
    let p = precProb [Pre] (Abs v t) t in
    cs (not b) "("++"\\"++show_ False v++"."++
    cs p "("++show_ True t++cs p ")"++cs (not b) ")"
  show_ b (Gen v t) =
    let p = precProb [Pre] (Gen v t) t in
    cs (not b) "("++"/\\"++show_ False v++"."++
    cs p "("++show_ True t++cs p ")"++cs (not b) ")"

  -- prefix
  show_ _ (InLR c e) | precProb [Pre] (InLR c e) e =
    "IN"++show c++" ("++show_ True e++")"
  show_ b (InLR c e) = "IN"++show c++" "++show_ b e

  -- apps
  show_ b (App t1 t2) =
    let lp = precProb [Post, LeftAss] (App t1 t2) t1
        rp = precProb [Pre] (App t1 t2) t2 in
    cs lp "("++show_ lp t1++cs lp ")"++" "++
    cs rp "("++show_ (rp||b) t2++cs rp ")"
  show_ b (Inst t (ty ::^ k)) =
    let lp = precProb [Post, LeftAss] (Inst t (ty ::^ k)) t
        rp = case prec ty of (_, Base) -> False
                             _ -> True in
    cs lp "("++show_ (lp||b) t++cs lp ")"++" "++
    cs rp "("++show_ (rp||b) (ty ::^ k)++cs rp ")"
```

```
  -- postfix
  show_ _ (Case inlr vl tl vr tr) =
    let p = precProb [Post, LeftAss] (Case inlr vl tl vr tr) inlr in
    cs p "("++ show_ p inlr++cs p ")"++
    "{"++show vl++"."++show_ True tl++", "++
    show vr++"."++show_ True tr++"}"
  show_ b (Lr c e) =
    let p = precProb [Post, LeftAss] (Lr c e) e in
    cs p "("++show_ (b||p) e++cs p ")"++" "++show c

  -- simple cases
  show_ _ (Var v) = v
  show_ _ (Pair t1 t2) = "<"++show_ True t1++", "++show_ True t2++">"

  -- ERROR
  show_ _ _ = "UntypedExp show_: no rule applies."

data UnkindedType
      = TypeVar String
      | TypeBinary Binary Type Type
      | TypeLambda Type Type
      | TypeApp Type Type
      | Forall Type Type
     deriving Eq
instance Show UnkindedType where
  show = show_ True -- first arg says whether term is in front of a closing
                    -- bracket to bracket \ . correctly

instance Prec UnkindedType where
  prec (TypeLambda _ _) = (1000, Pre)
  prec (Forall _ _) = (1000, Pre)
  prec (TypeBinary Function _ _) = (700, RightAss)
  prec (TypeBinary Sum _ _) = (600, LeftAss)
  prec (TypeBinary Prod _ _) = (500, LeftAss)
  prec (TypeApp _ _) = (100, LeftAss)
  prec (TypeVar _) = (0, Base)

precProbT ops out (inn ::^ t) =
     (fst (prec out) < fst (prec inn)
       && (snd (prec inn) `notElem` (ops `intersect` [Post, Pre])))
  || (fst (prec out) == fst (prec inn)
       && (snd (prec inn) `notElem` ops))
```

```
instance Show_ UnkindedType where
  show_ _ (TypeVar tv) = tv

  -- brackets if necessary
  show_ b (TypeLambda tv t) =
    cs (not b) "("++"\\"++show_ False tv++"."++show_ True t++cs (not b) ")"
  show_ b (Forall tv t) =
    cs (not b) "("++"&"++show_ False tv++"."++show_ True t++cs (not b) ")"

  -- binary infix operators + % ->
  show_ b (TypeBinary op t1 t2) =
    let lp = precProbT [Post, LeftAss] (TypeBinary op t1 t2) t1
        rp = precProbT [Pre, RightAss] (TypeBinary op t1 t2) t2 in
    cs lp "("++show_ lp t1++cs lp ")"++show op++
    cs rp "("++show_ (rp||b) t2++cs rp ")"

  -- apps
  show_ b (TypeApp t1 t2) =
    let lp = precProbT [Post, LeftAss] (TypeApp t1 t2) t1
        rp = precProbT [Pre, RightAss] (TypeApp t1 t2) t2 in
    cs lp "("++show_ lp t1++cs lp ")"++" "++
    cs rp "("++show_ (rp||b) t2++cs rp ")"

  -- base types
  show_ _ (TypeVar v) = v

  -- ERROR
  show_ _ _ = error "UnkindedType show_: no rule applies."

data Kind
      = KindProp
      | KindAbs Kind Kind
     deriving Eq
instance Show Kind where
  show KindProp = "*"
  show (KindAbs (KindAbs k11 k12) k2) =
         "("++show (KindAbs k11 k12)++")=>"++show k2
  show (KindAbs k1 k2) = show k1++"=>"++show k2

data Type = UnkindedType ::^ Maybe Kind deriving Eq

instance Show Type where
  show = show_ True
instance Show_ Type where
```

```
  show_ b (utype ::^ Nothing) = show_ b utype
  show_ b (utype ::^ Just kind) =
    let br = case prec utype of (_, Base) -> False
                                _ -> True in
    cs br "("++show_ (br||b) utype++cs br ")"++"^"++show kind

data Exp = UntypedExp ::: Maybe Type deriving Eq

instance Show Exp where
  show = show_ True
instance Show_ Exp where
  show_ b (uexp ::: Nothing) = show_ b uexp
  show_ b (uexp ::: Just (t ::^ k)) =
    show_ b uexp++":"++show_ b (t ::^ k)
```

3.1.2 Der Parser

```
-- this is a "happy" parser for system F_omega typed lambda
-- expressions with CASE and PAIR rules. All a la Church with
-- Maybe type assignments they can also be used to represent
-- Curry terms
-- $Id: ParseCPFw.y,v 1.12 2002/05/01 18:29:36 markus Exp $

{
module ParseCPFw where
import Char(isAlpha, isAlphaNum, isSpace, isUpper, isLower)
import CPFwDatatypes(Exp(..), Type(..), Kind(..),
                     UntypedExp(..), UnkindedType(..), Binary(..))

}

%name parser Exp
%tokentype { Token }

%token '\\'  { TokenAbs }
       '.'  { TokenDot }
       var  { TokenVar $$ }
       tvar { TokenTypVar $$ }
       ':'  { TokenColon }
       '&'  { TokenForall }
       '/\\' { TokenAlpha }
       '^'  { TokenUp }
       '+' { TokenPlus }
```

```
        '%' { TokenTimes }
        '->' { TokenArrow }
        'LR' { TokenLR $$ }
        'INLR' { TokenINLR $$ }
        '*'  { TokenStar }
        '=>' { TokenDoubleArrow }
        '('  { TokenOB }
        ')'  { TokenCB }
        '{'  { TokenCOB }
        '}'  { TokenCCB }
        ','  { TokenComma }
        '<'  { TokenROB }
        '>'  { TokenRCB }

-- precedence: earlier means lower

%nonassoc ':' -- type applicator

-- IN(L|R)
-- \ . lambda
-- /\ . Alpha

-- { } case
-- left: LR, church app, curry app

-- base:
-- Var
-- < , > pair
-- ( ) brackets

%left '%' -- % case type
%left '+' -- + pair type
%right '->' -- type abstractor

%right '=>' -- kind abstractor

%nonassoc '^' -- kind applicator

%monad { P } { thenP } { returnP }
%lexer { monadiclexer } { TokenEOF }

%%

-- EXP RULES
```

```
SimpleExp :: { Exp }
        : var { Var $1 ::: Nothing }
        | '<' Exp ',' Exp '>' { Pair $2 $4 ::: Nothing }
        | '(' Exp ')' { $2 }

-- left assoc. and postfix operators
AppExp :: { Exp }
        : SimpleExp { $1 }
        | AppExp SimpleExp { App $1 $2 ::: Nothing }
        | AppExp SimpleType { Inst $1 $2 ::: Nothing }
| AppExp 'LR' { Lr (read $2) $1 ::: Nothing }
        | AppExp '{' Exp '.' Exp ',' Exp '.' Exp '}'
          {% case ($3, $7) of
             (Var v1 ::: t1, Var v2 ::: t2) ->
               returnP (Case $1 (Var v1 ::: t1) $5 (Var v2 ::: t2) $9
                           ::: Nothing)
             (v1, v2) ->
               failP ("CASE, where "++show v1++" or "++show v2++
                      " are no variables.")
          }

-- expression with unbracketed prefix operators
NoBrakExp :: { Exp }
: AppExp { $1 }
| AppExp PreExp { App $1 $2 ::: Nothing }
| PreExp { $1 }

-- prefix operators
PreExp :: { Exp }
: 'INLR' NoBrakExp  { InLR (read $1) $2 ::: Nothing }
        | '\\' Exp '.' NoBrakExp
          {% case $2 of { Var v ::: t -> returnP (Abs (Var v ::: t)
                                                     $4
                                                    ::: Nothing)
                        ; e -> failP ("Abstraction of "++show e++
                                      "which is no variable.")
                        }
          }
        | '/\\' Type '.' NoBrakExp
          {% case $2 of { TypeVar v ::^ k -> returnP (Gen (TypeVar v ::^ k)
                                                         $4
                                                     ::: Nothing)
                        ; e -> failP ("GEN of "++show e++
```

```
                                        "which is no type variable.")
                        }
          }

Exp :: { Exp }
: NoBrakExp { $1 }
| Exp ':' Type {% case $1 of
                            uexp ::: Nothing -> returnP (uexp ::: Just $3)
                            _ -> failP "Double typing is not allowed" }


-- TYPE RULES

TypeVar :: { Type }
: tvar { TypeVar $1 ::^ Nothing }
| tvar '^' Kind { TypeVar $1 ::^ Just $3 }

SimpleType :: { Type }
: TypeVar { $1 }
| '(' Type ')' { $2 }
| '(' Type ')' '^' Kind
            {% case $2 of
                 utype ::^ Nothing -> returnP (utype ::^ Just $5)
                 _ -> failP "Double kinding is not allowed" }

AppType :: { Type }
: AppType SimpleType { TypeApp $1 $2 ::^ Nothing }
| SimpleType { $1 }

BinType :: { Type }
--: AppType { $1 }
: AppType '+' PreType { TypeBinary Sum $1 $3 ::^ Nothing }
| AppType '%' PreType { TypeBinary Prod $1 $3 ::^ Nothing }
| AppType '->' PreType { TypeBinary Function $1 $3 ::^ Nothing }

| BinType '+' PreType { TypeBinary Sum $1 $3 ::^ Nothing }
| BinType '%' PreType { TypeBinary Prod $1 $3 ::^ Nothing }
| BinType '->' PreType { TypeBinary Function $1 $3 ::^ Nothing }

| AppType '+' BinType { TypeBinary Sum $1 $3 ::^ Nothing }
| AppType '%' BinType { TypeBinary Prod $1 $3 ::^ Nothing }
| AppType '->' BinType { TypeBinary Function $1 $3 ::^ Nothing }

| BinType '+' BinType { TypeBinary Sum $1 $3 ::^ Nothing }
```

```
| BinType '%' BinType { TypeBinary Prod $1 $3 ::^ Nothing }
| BinType '->' BinType { TypeBinary Function $1 $3 ::^ Nothing }

| AppType '+' AppType { TypeBinary Sum $1 $3 ::^ Nothing }
| AppType '%' AppType { TypeBinary Prod $1 $3 ::^ Nothing }
| AppType '->' AppType { TypeBinary Function $1 $3 ::^ Nothing }

| BinType '+' AppType { TypeBinary Sum $1 $3 ::^ Nothing }
| BinType '%' AppType { TypeBinary Prod $1 $3 ::^ Nothing }
| BinType '->' AppType { TypeBinary Function $1 $3 ::^ Nothing }

-- types with unbracketed prefix operators
Type :: { Type }
: PreType { $1 }
| BinType { $1 }
| AppType { $1 }
| AppType PreType { TypeApp $1 $2 ::^ Nothing }

-- prefix operators
PreType :: { Type }
--: AppType { $1 }
--| BinType { $1 }
        : '\\' TypeVar '.' PreType { TypeLambda $2 $4 ::^ Nothing }
        | '&' TypeVar '.' PreType { Forall $2 $4 ::^ Nothing }
        | '\\' TypeVar '.' BinType { TypeLambda $2 $4 ::^ Nothing }
        | '&' TypeVar '.' BinType { Forall $2 $4 ::^ Nothing }
        | '\\' TypeVar '.' AppType { TypeLambda $2 $4 ::^ Nothing }
        | '&' TypeVar '.' AppType { Forall $2 $4 ::^ Nothing }

-- KIND RULES

Kind :: { Kind }
: '*' { KindProp }
| Kind '=>' Kind { KindAbs $1 $3 }
| '(' Kind ')' { $2 }

{
data Token
       = TokenAbs
       | TokenAlpha
       | TokenDot
       | TokenVar String
       | TokenTypVar String
       | TokenForall
```

```
        | TokenColon
        | TokenUp
        | TokenDoubleArrow
        | TokenStar
        | TokenOB
        | TokenCB
        | TokenPlus
        | TokenTimes
        | TokenArrow
        | TokenCOB
        | TokenCCB
        | TokenComma
        | TokenROB
        | TokenRCB
        | TokenLR String
        | TokenINLR String
        | TokenEOF
        | TokenFail
  deriving (Show, Eq)

-- a type to locate errors

type LineNumber = (Int, Int)
addLineNumber :: LineNumber->LineNumber->LineNumber
addLineNumber (x1,y1) (x2,y2) = (x1+x2, y1+y2)

-- the monad around the lexer to process errors

data ParseResult a = Ok a | Failed (String, LineNumber)

-- parse error handling

happyError :: P a
happyError = failP "Parse Error"

-- failP adds the line number, so i don't need this
-- getLineNumber :: P LineNumber
-- getLineNumber = \s l -> Ok l

-- the monadic lexer

type P a = String -> LineNumber -> ParseResult a
-- hand-made Monad P, since using a "real" Monad needs a real type, needs a
-- real constructor, creates really ugly functions with lots of meaningless
```

```
-- case constructs to eliminate these constructors

returnP :: a -> P a
returnP a = \s l -> Ok a

failP :: String -> P a
failP err = \ _ (x,y) -> Failed (err++" at line "++show y
                                 ++", column "++show x++".", (x,y))

thenP :: P a -> (a -> P b) -> P b
a `thenP` b = \s l -> case a s l of Ok r -> b r s l
                                    Failed err -> Failed err

monadiclexer :: (Token -> P a) -> P a
monadiclexer cont =
  \s l -> let (token, rest, pos)=lexer s l
          in if (token==TokenFail) then failP "Lexer error" rest pos
                                   else cont token rest pos

-- the real lexer

lexer :: String -> LineNumber -> (Token, String, LineNumber)
lexer [] (x,y) = (TokenEOF, "", (x,y))
lexer (c:cs) (x,y)
      | isSpace c =
          let (t,s, (xp,yp))=lexer cs (x,y)
           in (t,s, (if c=='\n' then (1,yp+1) else (xp+1,yp)))
      | isAlpha c =
          let (var, rest)=lexLetter (c:cs)
           in (var, rest, (length (c:cs)-length rest+x,y))
lexer ('\\':cs) (x,y) = (TokenAbs, cs, (x+1,y))
lexer ('/':'\\':cs) (x,y) = (TokenAlpha, cs, (x+2,y))
lexer ('.':cs) (x,y) = (TokenDot, cs, (x+1,y))
lexer (':':cs) (x,y) = (TokenColon, cs, (x+1,y))
lexer ('^':cs) (x,y) = (TokenUp, cs, (x+1,y))

lexer ('=':'>':cs) (x,y) = (TokenDoubleArrow, cs, (x+2,y))
lexer ('*':cs) (x,y) = (TokenStar, cs, (x+1,y))
lexer ('&':cs) (x,y) = (TokenForall, cs, (x+1,y))
lexer ('(':cs) (x,y) = (TokenOB, cs, (x+1,y))
lexer (')':cs) (x,y) = (TokenCB, cs, (x+1,y))

lexer ('+':cs) (x,y) = (TokenPlus, cs, (x+1,y))
lexer ('%':cs) (x,y) = (TokenTimes, cs, (x+1,y))
```

```
lexer ('-':'>':cs) (x,y) = (TokenArrow, cs, (x+2,y))
lexer ('{':cs) (x,y) = (TokenCOB, cs, (x+1,y))
lexer ('}':cs) (x,y) = (TokenCCB, cs, (x+1,y))
lexer (',':cs) (x,y) = (TokenComma, cs, (x+1,y))
lexer ('<':cs) (x,y) = (TokenROB, cs, (x+1,y))
lexer ('>':cs) (x,y) = (TokenRCB, cs, (x+1,y))


-- no rule applies
lexer s c = (TokenFail, s, c)

lexLetter cs =
  let (var, rest) = span isAlphaNum cs
      token = case var of
        "L" -> TokenLR "L"
        "R" -> TokenLR "R"
        "INL" -> TokenINLR "L"
        "INR" -> TokenINLR "R"

        v:_ | isUpper v -> TokenTypVar var
        v:_ | isLower v -> TokenVar var
  in (token, rest)

-- general functions

parserMonadic :: String -> ParseResult Exp
parserMonadic = \s -> parser s (1,1)

parserShortError :: String -> Exp
parserShortError s = case parser s (1,1) of Ok r -> r
                                            Failed (s,l) -> error s

marklocation s (x,y) =
  let ls=lines s
  in unlines (take y ls++[replicate (x-1) ' '++"^"]++drop y ls)

parserLongError :: String -> Exp
parserLongError s =
  case parser s (1,1) of
    Ok r -> r
    Failed (e,p) -> error (e++"\n"++marklocation s p)
parseCPFw = parserLongError
parse = parserLongError
```

```
}
```

3.1.3 Pretty-Printer

```
-- funtions to pretty print lambda terms with types as LaTeX source
-- $Id: PrintExplicit.hs,v 1.5 2002/05/01 18:29:36 markus Exp $

module PrintExplicit where
import ParseTypedLambdaExpressions(Exp(..), Type(..), Kind(..),
                                   UntypedExp(..), UnkindedType(..))

attribute separator string Nothing = string
attribute separator string (Just t) = "("++string++separator++show t++")"

class Explicit a where
  explicit :: a -> String

instance Explicit Exp where
  explicit (Abs v t ::: ty) =
    attribute ":" ("(\\"++explicit v++"."++explicit t++")") ty
  explicit (App t1 t2 ::: ty) =
    attribute ":" ("("++explicit t1++" "++explicit t2++")") ty
  explicit (Var v ::: ty) =
    attribute ":" v ty

instance Explicit Kind where
  explicit KindProp = "*"
  explicit (KindAbs k1 k2) = "("++explicit k1++"=>"++explicit k2++")"

instance Explicit Type where
  explicit (TypeAbs t1 t2 ::^ k) =
    attribute "^" ("("++explicit t1++"->"++explicit t2++")") k
  explicit (TypeApp t1 t2 ::^ k) =
    attribute "^"  ("("++explicit t1++" "++explicit t2++")") k
  explicit (Forall (TypeVar tv ::^ ktv) t ::^ k) =
    attribute "^" ("(&"++explicit (TypeVar tv ::^ ktv)++"."++explicit t++")") k
  explicit (TypeVar tv ::^ k) =
    attribute "^" tv k

-- funtions to pretty print lambda terms with types as LaTeX source
-- $Id: PrintLaTeX.hs,v 1.8 2002/05/08 02:32:09 markus Exp $

module PrintLaTeX where
```

```
import CPFwDatatypes(Exp(..), Type(..), UntypedExp(..), UnkindedType(..),
                     Kind(..),
                     Binary (..),
                     Prec(..), precProb, precProbT, OpStyle(..),
                     cs)

import Char(isAlphaNum)

latexVar v =
  let s=fst $ span isAlphaNum v in if s==v then s else s++"..."
--  let cs=drop 1 (show v) in "\\textrm{ "++take (length cs-1) cs++" }"

class LaTeX a where
  latex :: a -> String
  latex = latex_ True
  latex_ :: Bool -> a -> String

instance LaTeX UntypedExp where
  -- dot structures (prefix), lowest prec so no testing neccesary
  latex_ b (Abs v t) =
    let p = precProb [Pre] (Abs v t) t in
    cs (not b) "("++"\\lambda "++latex_ False v++"."++
    cs p "("++latex_ True t++cs p ")"++cs (not b) ")"
  latex_ b (Gen v t) =
    let p = precProb [Pre] (Gen v t) t in
    cs (not b) "("++"\\Lambda "++latex_ False v++"."++
    cs p "("++latex_ True t++cs p ")"++cs (not b) ")"

  -- prefix
  latex_ _ (InLR c e) | precProb [Pre] (InLR c e) e =
    "IN"++show c++" ("++latex_ True e++")"
  latex_ b (InLR c e) = "IN"++show c++" "++latex_ b e

  -- apps
  latex_ b (App t1 t2) =
    let lp = precProb [Post, LeftAss] (App t1 t2) t1
        rp = precProb [Pre] (App t1 t2) t2 in
    cs lp "("++latex_ lp t1++cs lp ")"++" "++
    cs rp "("++latex_ (rp||b) t2++cs rp ")"
  latex_ b (Inst t (ty ::^ k)) =
    let lp = precProb [Post, LeftAss] (Inst t (ty ::^ k)) t
        rp = case prec ty of (_, Base) -> False
                             _ -> True in
    cs lp "("++latex_ (lp||b) t++cs lp ")"++" "++
```

```
    cs rp "("++latex_ (rp||b) (ty ::^ k)++cs rp ")"

  -- postfix
  latex_ _ (Case inlr vl tl vr tr) =
    let p = precProb [Post, LeftAss] (Case inlr vl tl vr tr) inlr in
    cs p "("++ latex_ p inlr++cs p ")"++
    "( "++show vl++" . "++latex_ True tl++" , "++
    show vr++" . "++latex_ True tr++" )"
{- latex is unable to typeset bracketed expressions reasonable
    cs p "("++ latex_ p inlr++cs p ")"++
    "\\left( "++show vl++". \\right. "++latex_ True tl++", "++
    show vr++" . \\left. "++latex_ True tr++" \\right)"
-}
  latex_ b (Lr c e) =
    let p = precProb [Post, LeftAss] (Lr c e) e in
    cs p "("++latex_ (b||p) e++cs p ")"++" "++show c

  -- simple cases
  latex_ _ (Var v) = latexVar v
  latex_ _ (Pair t1 t2) = " < "++latex_ True t1++", "++
                          latex_ True t2++" > "
{- latex is unable to typeset bracketed expressions reasonable
  latex_ _ (Pair t1 t2) = " \\left< "++latex_ True t1++", \\right. \\left. "++
                          latex_ True t2++" \\right> "
-}

  -- ERROR
  latex_ _ _ = "UntypedExp latex_: no rule applies."

instance LaTeX Exp where
  latex_ b (e ::: Nothing) = latex_ b e
  latex_ b (e ::: Just t) = latex_ b e++":"++latex_ b t

instance LaTeX Kind where
  latex_ _ KindProp = "*"
  latex_ _ (KindAbs (KindAbs k11 k12) k2) =
    "("++latex (KindAbs k11 k12)++") \\Rightarrow "++latex k2
  latex_ _ (KindAbs k1 k2) = latex k1++" \\Rightarrow "++latex k2

instance LaTeX Binary where
  latex_ _ Sum = " + "
  latex_ _ Prod = " \\times "
  latex_ _ Function = " \\rightarrow "
```

```
instance LaTeX UnkindedType where
  latex_ _ (TypeVar tv) = latexVar tv

  -- brackets if necessary
  latex_ b (TypeLambda tv t) =
    cs (not b) "("++" \\lambda "++latex_ False tv++"."++latex_ True t++cs (not b) ")"
  latex_ b (Forall tv t) =
    cs (not b) "("++" \\forall "++latex_ False tv++"."++latex_ True t++cs (not b) ")"

  -- binary infix operators + % ->
  latex_ b (TypeBinary op t1 t2) =
    let lp = precProbT [Post, LeftAss] (TypeBinary op t1 t2) t1
        rp = precProbT [Pre, RightAss] (TypeBinary op t1 t2) t2 in
    cs lp "("++latex_ lp t1++cs lp ")"++latex op++
    cs rp "("++latex_ (rp||b) t2++cs rp ")"

  -- apps
  latex_ b (TypeApp t1 t2) =
    let lp = precProbT [Post, LeftAss] (TypeApp t1 t2) t1
        rp = precProbT [Pre, RightAss] (TypeApp t1 t2) t2 in
    cs lp "("++latex_ lp t1++cs lp ")"++" "++
    cs rp "("++latex_ (rp||b) t2++cs rp ")"

  -- base types
  latex_ _ (TypeVar tv) = latexVar tv

  -- ERROR
  latex_ _ _ = error "UnkindedType latex_: no rule applies."

instance LaTeX Type where
  latex_ b (t ::^ Nothing) = latex_ b t
  latex_ b (TypeVar v ::^ Just k) = latex_ b (TypeVar v)++"^{"++latex k++"}"
  latex_ b (t ::^ Just k) = "("++latex_ True t++")^{"++latex k++"}"
```

3.1.4 Die Verarbeitung der Bäume

```
-- Implementing funtions for lambda terms with types
-- $Id: ProcessCPFw.hs,v 1.18 2002/05/07 20:09:44 markus Exp $

module ProcessCPFw where
import CPFwDatatypes(Exp(..), Type(..), UntypedExp(..), UnkindedType(..),
                     Kind(..), Binary(..), LR(..), ty, jty, con, kd, jkd)
```

```
import List(union, unionBy, intersect, (\\), elemIndex, delete, nubBy, nub,
            find)
import Maybe(mapMaybe, fromJust, isNothing, isJust, maybeToList,
             catMaybes, fromMaybe)
import Char(isDigit)
import Monad(liftM2, mplus)

-- class Lambda with methods to ease the implementation of methods from the
-- lambda calculus theory

class Eq a=>Lambda a where
  -- a more uniform representation of trees with bound variables:
  -- a list of pairs of a bound variable (or Nothing if nothing is bound) and
  -- the term it is abstracted out
  destr :: a->[(Maybe a, a)]
  constr :: a->[(Maybe a, a)]->a
  errorvar :: String -> a
  sameType :: a->a->Bool
  rmType :: a->a
  freshenVar :: a->a
  isLambda :: a->Bool
  isApp :: a->Bool
  redEx :: a->Maybe a

  match :: [a] -> a -> a -> a
  annotateType :: [a] -> a -> a
  annotateInner :: [a] -> a -> a

-- comparing two types using alpha equality
infix 4 ==+
(==+) :: (Lambda a)=>a->a->Bool

a ==+ b = let l=alphaUnify (destr a) (destr b)
              join (b1, l1) (b2, l2) = (b1&&b2, l1++l2)
              alphaUnify :: (Lambda a)=>[(Maybe a, a)] -> [(Maybe a, a)] -> (Bool, [(Maybe
              alphaUnify [] [] = (True, [])
              alphaUnify ((Nothing, tl):rl) ((Nothing, tr):rr) =
                 join (tl ==+ tr, [(Nothing, tl)]) (alphaUnify rl rr)
              alphaUnify ((vl, tl):rl) ((vr, tr):rr) =
                 join (tl ==+ snd (alpha (vr, tr) (fromJust vl)), [(vl, tl)]) (alphaUnify
              alphaUnify _ _ = (False, [])
          in fst l && constr a (snd l) == constr b (snd l)
```

```
instance Lambda Exp where

  destr (Var v ::: ty) =
    []
  destr (Abs (Var v ::: ty) tm ::: t) =
    [(Just (Var v ::: ty), tm)]
  destr (Inst e ty ::: t) =
    [(Nothing, e)]
  destr (Gen v e ::: t) =
    [(Nothing, e)]
  destr (App e1 e2 ::: t) =
    [(Nothing, e1), (Nothing, e2)]
  destr (Case e (Var v1 ::: ty1) t1 (Var v2 ::: ty2) t2 ::: t) =
    [(Nothing, e), (Just (Var v1 ::: ty1), t1), (Just (Var v2 ::: ty2), t2)]
  destr (InLR lr e ::: t) =
    [(Nothing, e)]
  destr (Pair e1 e2 ::: t) =
    [(Nothing, e1), (Nothing, e2)]
  destr (Lr lr e ::: t) =
    [(Nothing, e)]
  destr a = error ("destr: no rule applies to "++show a++".")

  constr (Var v ::: ty) [] =
    (Var v ::: ty)
  constr (Abs _ _ ::: t) [(Just (Var v ::: ty), tm)] =
    Abs (Var v ::: ty) tm ::: t
  constr (Inst _ ty ::: t) [(Nothing, e)]  =
    Inst e ty ::: t
  constr (Gen v _ ::: t) [(Nothing, e)]  =
    Gen v e ::: t
  constr (App _ _ ::: t) [(Nothing, e1), (Nothing, e2)]  =
    App e1 e2 ::: t
  constr (Case _ _ _ _ _ ::: t)
         [(Nothing, e), (Just (Var v1 ::: ty1), t1), (Just (Var v2 ::: ty2), t2)]  =
    Case e (Var v1 ::: ty1) t1 (Var v2 ::: ty2) t2 ::: t
  constr (InLR lr _ ::: t) [(Nothing, e)]  =
    InLR lr e ::: t
  constr (Pair _ _ ::: t) [(Nothing, e1), (Nothing, e2)]  =
    Pair e1 e2 ::: t
  constr (Lr lr _ ::: t) [(Nothing, e)]  =
    Lr lr e ::: t
  constr a b = errorvar ("constr: no rule applies to wrap "++show a++" around "++
                         show b++".")
```

```
errorvar s = Var s ::: Nothing

freshenVar (Var v ::: typ) =
  let (n,r)=span isDigit (reverse v)
  in Var (reverse r++show (read ("0"++(reverse n)) + 1)) ::: typ

sameType e1 e2 = ty e1 == ty e2

rmType (tm ::: _) = tm ::: Nothing

isLambda (Abs _ _ ::: _) = True
isLambda _ = False

isApp (App _ _ ::: _) = True
isApp _ = False

redEx (App (Abs v t ::: atyp) t2 ::: typ) =
  Just (subst [(v, t2)] t)
redEx (Case (InLR lr r ::: intyp) vl tl vr tr ::: typ) =
  let lrf=if lr==L then fst else snd
  in Just (subst [(lrf (vl,vr), r)] (lrf (tl, tr)))
redEx (Lr lr (Pair t1 t2 ::: ptyp) ::: typ) =
  Just (if lr==L then t1 else t2)
redEx _ = Nothing

match env (e1 ::: Nothing) (e2 ::: t2) =
  (e2 ::: fmap (annotateFull_ (freetv env)) t2)
match env (e1 ::: t1) (e2 ::: t2) | matchingT (betaReduction $ fromJust t1)
                                              (betaReduction $ fromJust t2) =
  (e2 ::: fmap (annotateFull_ (freetv env).betaReduction) t2)
match _ (e1 ::: t1) (e2 ::: t2) =
  errorvar ("Can not match "++show (e1 ::: fmap betaReduction t1)++
            " with "++show (e2 ::: fmap betaReduction t2)++".")

annotateInner env (Inst e t ::: typ) =
  case jty e of
    Forall v r ::^ Just KindProp ->
      match env (Inst e t ::: typ)
            (Inst e t ::: Just (subst [(v,t)] r))
    _ -> errorvar ("Can not INST a type "++show t++" without a & "++
                   "at the front of "++show (jty e)++".")
annotateInner env (Abs v e ::: typ) =
  match env (Abs v e ::: typ)
        (Abs v e ::: Just (TypeBinary Function
```

```
                                        (typeToProp (jty v))
                                        (typeToProp (jty e))
                                ::^ Just KindProp))
annotateInner env (Gen t e ::: typ) =
  case ty e of
    Nothing -> errorvar ("Untyped term in Lambda (/\\): "++show e++".")
    Just tye -> match env (Gen t e ::: typ)
                          (Gen t e ::: Just (Forall t (typeToProp tye)
                                                  ::^ Just KindProp))
annotateInner env (App e1 e2 ::: typ) =
  case jty e1 of
    TypeBinary Function v t ::^ k ->
      if v==jty e2 then match env (App e1 e2 ::: typ)
                                (App e1 e2 ::: Just t)
                   else errorvar ("Cannot apply "++show e1++" $ "++show e2++
                                  ". Type mismatch.")
    _ -> errorvar ("Cannot apply "++show e1++" $ "++show e2++". No APP type.")
annotateInner env (Lr lr e ::: typ) =
  case jty e of
    TypeBinary Prod tl tr ::^ Just KindProp ->
      match env (Lr lr e ::: typ)
            (Lr lr e ::: Just (case lr of L -> tl; R -> tr))
    t -> errorvar ("Can not LR a type "++show t++" which is not a product.")
annotateInner env (Pair e1 e2 ::: typ) =
  match env (Pair e1 e2 ::: typ)
        (Pair e1 e2 ::: Just (TypeBinary Prod (jty e1)
                                              (jty e2)
                                 ::^ Just KindProp))
annotateInner env (Case ei v1 e1 v2 e2 ::: typ) =
  case jty ei of
    TypeBinary Sum tl tr ::^ k ->
      if k /= Just KindProp then errorvar "Case disjunction not of type *." else
      if tl /= jty v1 then errorvar "Case type mismatch in left argument." else
      if tr /= jty v2 then errorvar "Case type mismatch in right argument." else
      if ty e1 /= ty e2 then errorvar "Case type mismatch in fuction arguments." else
        match env (Case ei v1 e1 v2 e2 ::: typ) (Case ei v1 e1 v2 e2 ::: ty e1)
    t -> errorvar ("Can not Case an expression with a type "++show t++
                   " which is not a sum.")
annotateInner env (InLR lr e ::: typ) =
  case typ of
    Nothing -> errorvar ("Can not type a InLR "++show (InLR lr e ::: typ)++
                         " without annotation.")
    Just (TypeBinary Sum lt rt ::^ k) ->
      match env (InLR lr e ::: typ)
```

```
              (InLR lr e ::: Just (case lr of L -> TypeBinary Sum (jty e) rt
                                             R -> TypeBinary Sum lt (jty e)
                                    ::^ toProp k))
       Just t -> errorvar ("Type "++show t++"doesn't match InLR "++
                           show (InLR lr e ::: typ)++".")

  annotateInner env e = errorvar("annotateInner: Can not annotate "++
                                 show e++" under env "++show env++".")

  annotateType env (e ::: Just t) =
    e ::: Just (annotateFull_ (freetv env) t) -- annotate variable binders
  annotateType env te =
    errorvar ("annotateType(Exp): cannot annotate empty type in: "++
              show te++".")

instance Lambda Type where

  destr (TypeVar v ::^ ty) =
    []
  destr (TypeLambda (TypeVar v ::^ ty) tm ::^ t) =
    [(Just (TypeVar v ::^ ty), tm)]
  destr (Forall (TypeVar v ::^ ty) tm ::^ t) =
    [(Just (TypeVar v ::^ ty), tm)]
  destr (TypeApp e1 e2 ::^ t) =
    [(Nothing, e1), (Nothing, e2)]
  destr (TypeBinary _ e1 e2 ::^ t) =
    [(Nothing, e1), (Nothing, e2)]
  destr a = error ("destr: no rule applies to "++show a++".")

  constr (TypeVar v ::^ ty) [] =
    TypeVar v ::^ ty
  constr (TypeLambda _ _ ::^ t) [(Just (TypeVar v ::^ ty), tm)] =
    TypeLambda (TypeVar v ::^ ty) tm ::^ t
  constr (Forall _ _ ::^ t) [(Just (TypeVar v ::^ ty), tm)] =
    Forall (TypeVar v ::^ ty) tm ::^ t
  constr (TypeApp _ _ ::^ t) [(Nothing, e1), (Nothing, e2)]  =
    TypeApp e1 e2 ::^ t
  constr (TypeBinary k _ _ ::^ t) [(Nothing, e1), (Nothing, e2)]  =
    TypeBinary k e1 e2 ::^ t
  constr a b = errorvar ("constr: no rule applies to wrap "++show a++" around "++
                         show b++".")

  errorvar s = TypeVar s ::^ Nothing
```

```
freshenVar (TypeVar v ::^ k) =
  let (n,r)=span isDigit (reverse v)
  in TypeVar (reverse r++show (read ("0"++(reverse n)) + 1)) ::^ k

sameType e1 e2 = kd e1 == kd e2

rmType (tm ::^ _) = tm ::^ Nothing

isLambda (TypeLambda _ _ ::^ _) = True
isLambda _ = False

isApp (TypeApp _ _ ::^ _) = True
isApp _ = False

redEx (TypeApp (TypeLambda v t ::^ ak) t2 ::^ tk) =
  Just (subst [(v, t2)] t)
redEx _ = Nothing

match _ (e1 ::^ Nothing) (e2 ::^ Nothing) =
  errorvar ("Can not match "++show e1++" with "++show e2++
            " having both no kinds.")
match _ (e1 ::^ Nothing) (e2 ::^ t2) = (e2 ::^ t2)
match _ (e1 ::^ t1) (e2 ::^ t2) | t1 == t2 = (e2 ::^ t2)
match _ e1 e2 =
  errorvar ("Can not match "++show e1++" with "++show e2++".")

annotateInner env (TypeBinary q lt rt ::^ k) =
  match env (TypeBinary q lt rt ::^ k)
        (TypeBinary q (typeToProp lt) (typeToProp rt) ::^ toProp k)
annotateInner env (Forall v t ::^ k) =
  match env (Forall v t ::^ k)
        (Forall v (typeToProp t) ::^ toProp k)
annotateInner env (TypeLambda v t ::^ k) =
  match env (TypeLambda v t ::^ k)
        (TypeLambda v t ::^ Just (KindAbs (jkd v) (jkd t)))
annotateInner env (TypeApp t1 t2 ::^ k) =
  case jkd t1 of
    KindAbs k1 k2 | k1 == jkd t2 ->
      match env (TypeApp t1 t2 ::^ k)
            (TypeApp t1 t2 ::^ Just k2)
    kw -> errorvar ("Kind mismatch TypeApp: "++show kw++" $ "++
                    show (jkd t2)++".")

annotateInner env t = errorvar ("annotateInner: no rule applies. "++show t++".")
```

```
  annotateType tenv t = annotateFull_ [] t -- used for variable binders

matchingT (e1 ::^ Nothing) (e2 ::^ t2) = True
matchingT (e1 ::^ t1) (e2 ::^ t2) =
  t1 == t2 {- && all (\(a,b)->matchingT a b)  (zip  ((map snd (destr e1)))
                                                   (map snd (destr e2))) -}

freetv [] = []
freetv (e:es) = freevar (jty e)++freetv es

isVar :: Lambda a => a->Bool
isVar = null . destr

isBinder t = (do (b,_)<-destr t; maybeToList b) /= []

var :: Lambda a=>a->[a]
var e =
  if isVar e then [e]
             else nubBy sameName (do (b,t)<-destr e
                                     unionBy sameName (maybeToList b) (var t))

freevar :: Lambda a=>a->[a]
freevar e =
  if isVar e then [e]
             else nubBy sameName
                        (do (b,t)<-destr e
                            (filter (\v->Just (rmType v) /= fmap rmType b)
                                    (freevar t)))

boundvar :: Lambda a=>a->[a]
boundvar e = nubBy sameName  (do (b,t)<-destr e
                                 unionBy sameName (maybeToList b) (boundvar t))

-- subterms

directSubterms :: Lambda a=>a->[a]
directSubterms exp = map snd $ destr exp

subterms :: Lambda a=>a->[a]
subterms exp = subterms_ [exp] where
                 subterms_ [] =[]
                 subterms_ (t:ts) = t:subterms_ (directSubterms t++ts)
```

```
-- no capture avoidance
capsubst :: Lambda a=>[(a, a)]->a->a
capsubst s e | isVar e = case e `lookup` s of Nothing -> e
                                              Just v -> v
capsubst s e = constr e (do (b,t)<-destr e
                            let n=case b of Just v -> (filter ((v==).fst) s)
                                            Nothing -> s
                            [(b, capsubst s t)])

-- change names of bound variables
alpha :: Lambda a => (Maybe a, a) -> a -> (Maybe a, a)
alpha (Just vold, t) vnew
  | not (sameType vold vnew) = error ("alpha subsitutes only "++
                                      "variables of the same type.")
  | not (isVar vnew) = error ("alpha needs a new variable as "++
                              "second argument.")
  | vnew `elem` freevar t\\[vold] = error "alpha name clash."
  | True = (Just vnew, capsubst [(vold, vnew)] t)
alpha _ _ = error ("alpha: needs a Just binder pair as "++
                   "first and a new variable as second argument.")

subst :: Lambda a => [(a, a)] -> a -> a -- [(variable, term to insert)], target ->result
                                        -- parallel substitution of variables

subst s t | isVar t = fromMaybe t (t `lookup` s)
subst s t =
  constr t (do (mv, tm)<-destr t
               case mv of Nothing -> [(Nothing, subst s tm)]
                          Just v -> if (freevar t `intersect` map fst s) /= []
                                       -- no substitution, no capture
                                       && (v `elem` (do (sv,st)<-s; sv:freevar st))
                                       -- capture danger
                                    then -- rename
                                      let fv=freshenVar v
                                          asub=alpha (mv, tm) fv
                                      in destr (subst s (constr t [asub]))
                                    else [(mv, subst s tm)]

           )

-- betaReduce :: Lambda a => a -> Maybe a

betaReduce t | isApp t && (isLambda (snd (destr t !! 0))) =
  let arg=snd (destr t !! 1)
```

```
      [(Just var, fct)]=destr (snd (destr t !! 0))
  in (Just (subst [(var, arg)] fct))

-- betaReduce t = error ("BLOCK error: "++show t)

betaReduce t | isVar t = Nothing

betaReduce t =  fmap (constr t) (leftmost (destr t)) where
--  leftmost :: Lambda a => [(Maybe a, a)]->Maybe [(Maybe a, a)]
  leftmost [] = Nothing
  leftmost (p:ps) = case betaReduce (snd p) of
    Nothing -> fmap (p:) (leftmost ps)
    Just nt -> Just ((fst p, nt):ps)


-- reduce using all reduction rules avaliable in the calculus of "Lambda"
-- this is governed by the redEx Lambda class method

fullReduce :: Lambda a => a -> Maybe a
fullReduce t | isJust (redEx t) = redEx t
fullReduce t = fmap (constr t) (leftmost (destr t)) where
  leftmost [] = Nothing
  leftmost (p:ps) =
    case fullReduce (snd p) of
      Nothing -> fmap (p:) (leftmost ps)
      Just nt -> Just ((fst p, nt):ps)

-- reduce until no reduction applies or term stops changing
reduction :: Lambda a => (a->Maybe a)->a->a
reduction r t =
  case r t of Nothing -> t
              Just nt -> if t==nt then t
                                  else reduction r nt

betaReduction :: Lambda a => a -> a
betaReduction = reduction betaReduce

fullReduction :: Lambda a => a -> a
fullReduction = reduction fullReduce

-- a history of recorded reductions

historyReduction r t =
  case r t of Nothing -> []
```

```
              Just nt -> if t==nt then []
                                  else nt:historyReduction r nt

historyBetaReduction t = t:historyReduction betaReduce t

historyFullReduction t = t:historyReduction fullReduce t

-- helper

noType t = sameType t (rmType t)
sameName v1 v2 = rmType v1 == rmType v2
addVar v l = if noType v then [errorvar ("Cannot add untyped variable "++show v++
                                         " to "++show l++".")]
                         else nubBy sameName (v:l)

freetypevarInEnv env =
  nub (do _ ::: t<-env
          case t of
            Nothing -> [errorvar ("freetypevarInEnv: "++
                                  show env++" contains untyped elements.")]
            Just typ -> let fv=freevar (con typ ::^ toProp (kd typ)) in
                        if any noType fv
                          then [errorvar ("freetypevarInEnv: "++show env++
                                          " contains unkinded type variables.")]
                          else fv
      )

toProp Nothing = Just KindProp
toProp (Just KindProp) = Just KindProp
toProp k = error ("toProp: Can not convert kind "++show k++" to kind Prop.")

rmProp Nothing = Nothing
rmProp (Just KindProp) = Nothing
rmProp k = error ("rmProp: Kind "++show k++" is no Prop.")

typeToProp (t ::^ k) = t ::^ toProp k

-- strips

strip t | isVar t = rmType t
strip t = rmType ((constr t) (do (v,st)<-destr t
                                 [(fmap rmType v, strip st)]))

stripChurch t = stripChurch_ [] t
```

```
stripChurch_ env t | isVar t =
  if t `elem` env then rmType t
                  else t -- stripType t
stripChurch_ env t =
    rmType ((constr t)
      (do (v, st)<-destr t
          case v of
            Just var -> [(Just (stripChurch_ [] var),
                          stripChurch_ (addVar var env) st)]
            Nothing -> [(v, stripChurch_ env st)])
    )

-- annotate

-- annotateFull annotates all inner nodes on a tree with annotated leafes
-- using a bottom up algorithm
annotateFull :: (Lambda a, Show a) => a -> a
annotateFull t = annotateFull_ (freevar t) t

annotateFull_ :: (Lambda a, Show a) => [a] -> a -> a
annotateFull_ env t | any noType env =
  errorvar ("annotateFull_: all variables in environment "++
            show env++" have to be typed. Problem:"++
            show (filter noType env)++".")
annotateFull_ env t | isVar t && noType t =
  case t `lookup` (zip (map rmType env) env) of
    Nothing -> errorvar ("annotateFull_: (here) free variable "++show t++
                         " not found in environment: "++show env++".")
    Just v -> v
annotateFull_ env t | isVar t =
  case find (sameName t) env of
    Nothing -> if rmType t == t
                  then errorvar ("annotateFull_: free variables must"++
                                 "be annotated. Env: "++show env++
                                 " Var: "++show t++".")
                  else t
    Just envvar -> if rmType t == t || envvar == t
                      then envvar
                      else errorvar ("annotateFull_: inner variable "++
                                     "not correctly annotated. EnvVar: "++
                                     show envvar++" Var: "++show t++".")

annotateFull_ env t =
```

```
  annotateInner env (constr t
    (do (v,st)<-destr t
        case v of
          Nothing -> [(v, annotateFull_ env st)]
          Just var -> if (noType var)
                        then [(Nothing,
                               errorvar ("annotateFull_: need annotated "++
                                         "abstracted variables. Problem"++show v++
                                         " in "++show t++"."))]
                        else let varan = annotateType env var in
                             [(Just varan,
                               annotateFull_ (addVar varan env) st)]
    ))

annotateChurch :: (Lambda a, Show a)=>a -> a
annotateChurch = stripChurch . annotateFull

-- general type checking function

createType :: Exp -> [Type] -> Type
createType (Abs v _ ::: _) [t] =
  TypeBinary Function (jty v) t ::^ Just KindProp
createType (Inst _ t ::: _) [Forall v r ::^ Just KindProp] =
  betaReduction $ subst [(v,t)] r
createType (Gen v e ::: _) [t] =
  if v `elem` (freetv (freevar e))
    then errorvar ("\nERROR: Illegal Generalisation. TVar:"++
                   show v++" free in "++show e++".\n"++
                   "freevar e: "++show (freevar e)++"\n"++
                   "freetv (freevar e): "++show(freetv (freevar e))++"\n---\n")
    else Forall v t ::^ Just KindProp
createType (App _ _ ::: _) [TypeBinary Function t1 t2  ::^ Just KindProp, t3] =
  t2 -- t1 == t3 not checked
createType (Var _ ::: t) [] =
  fromJust t
createType (Case c vl el vr er ::: t) [tc,tel,ter] =
  ter -- not checked:
      -- con (jty c) == TypeBinary Sum (jty vl) (jty vr) &&
      -- jty el == jty er
createType (InLR lr _ ::: Just (TypeBinary Sum tl tr  ::^ Just KindProp)) [te] =
  case lr of L-> TypeBinary Sum te tr  ::^ Just KindProp
             R-> TypeBinary Sum tl te  ::^ Just KindProp
createType (Pair e1 e2 ::: t) [t1,t2] =
  TypeBinary Prod t1 t2 ::^ Just KindProp
```

```
createType (Lr lr _ ::: t) [TypeBinary Prod tl tr ::^ Just KindProp] =
  case lr of L->tl; R->tr
createType e ts = errorvar ("ERROR: Can not create type. No rule applies for "++
                            show (e, ts)) -- ERROR

checkType exp = map checkType_ (subterms exp)

checkType_ e = matchType e (jty e) ((createType e) (map jty (directSubterms e)))

matchType e t1 t2 = (e, t1, t2, t1 ==+ t2)

-- check whether a Church term is completely and correctly annotated

checkChurch :: Exp -> Maybe String -- Nothing means OK, Just Error

checkChurch (Abs e1 e2 ::: t) =
  checkChurch e1 `mplus` checkChurch e2 `mplus`
    if (TypeBinary Function (jty e1) (jty e2) == con (fromJust t))
      then Just ("Abs mismatch:"++show (jty e1)++"->"++
                 show(jty e2)++"!="++show (con (fromJust t))++".")
      else Nothing
checkChurch (Inst e t ::: ot) =
  checkChurch e  `mplus`
    case jty e of
    Forall v r ::^ Just KindProp ->
      if Just (betaReduction $ subst [(v,t)] r) == ot
        then Nothing
        else Just "INST type mismatch"
    n ->
      Just ("Can't INST a type "++show n++
            "which is not a Forall.")
checkChurch (Gen v e ::: t) =
  checkChurch e `mplus`
  if con (fromJust t) == Forall v (jty e)
    then Nothing
    else Just "Gen fail."
checkChurch (App e1 e2 ::: t) =
  checkChurch e1 `mplus` checkChurch e2 `mplus`
  if (TypeBinary Function (jty e2) (fromJust t) == con (jty e1))
    then Nothing
    else Just "App fail."
checkChurch (Var s ::: t) = Nothing -- use an environment?
checkChurch (Case c vl el vr er ::: t) =
  checkChurch c `mplus` checkChurch vl `mplus` checkChurch el
```

```
  `mplus`  checkChurch vr `mplus`  checkChurch er `mplus`
  if con (jty c) == TypeBinary Sum (jty vl) (jty vr) &&
     jty el == jty er && jty er == fromJust t
  then Nothing
  else Just "Case fail."
checkChurch (InLR lr e ::: t) =
  checkChurch e `mplus`
  case con (fromJust t) of
    TypeBinary Sum tl tr ->
      if (case lr of L->tl; R->tr) == jty e
        then Nothing
        else Just "InLR fail."
    n -> Just ("Type of InLR has to be a Sum.")
checkChurch (Pair e1 e2 ::: t) =
  checkChurch e1 `mplus` checkChurch e2 `mplus`
  if TypeBinary Prod (jty e1) (jty e2) == con (fromJust t)
    then Nothing
    else Just "Pair fail."
checkChurch (Lr lr e ::: t) =
  checkChurch e `mplus`
  case con (jty e) of
    TypeBinary Prod tl tr ->
      if (case lr of L->tl; R->tr) == fromJust t
        then Nothing
        else Just "LR fail."
    n -> Just ("LR application needs a Prod type:"++show n++".")
```

3.1.5 Die Beispiele

```
-- Examples to test the type checker
-- $Id: Examples.hs,v 1.21 2002/05/07 21:26:29 markus Exp $

module Examples where

import CPFwDatatypes(Exp(..), Type(..), UntypedExp(..), UnkindedType(..),
                     Kind(..))
import ParseCPFw(parse)

import ProcessCPFw

--import PrintExplicit(Explicit(..))
import PrintLaTeX(LaTeX(..))
```

```
import Maybe(fromMaybe)

-- adapted show for lists

brshow :: Show a => [a] -> String
brshow [] = "[]"
brshow (e:es) = "["++foldl (\s e->s++",\n "++show e) (show e) es++"\n]"

-- combinators s,k,i from combinator logic
s = parse("\\x.\\y.\\z.(x z)(y z)")
k = parse("\\x.\\y.x")
i = parse("\\x.x")

skk = (((s `App` k) ::: Nothing)
           `App` k) ::: Nothing

-- Church numerals
church :: Int->Exp
church n = parse("\\f.\\x."++church_ n)
church_ :: Int->String
church_ 0 = "x"
church_ n = "f("++church_ (n-1)++")"

tchurch n = parse("(\\f:(A^*->A^*)^*.\\x:A^*."++tchurch_ n
                        ++"):((A^*->A^*)^*->(A^*->A^*)^*)^*")
tchurch_ :: Int->String
tchurch_ 0 = "(x:A^*)"
tchurch_ n = "((f:(A^*->A^*)^*)"++tchurch_ (n-1)++":A^*)"

-- Urzyczyn's 2-chains
urzy n = ((urzy_ n) `App` k) ::: Nothing
urzy_ 0 = church 2
urzy_ n = ((urzy_ (n-1)) `App` (church 2)) ::: Nothing

-- Urzyczyn's 22K p. 338

typedtwotwok="("++show (church 2)++")("++show (church 2)++
  ":&B.(&A.A->(B A))->(&A.A->(B(B A)))"++")("++show k++")"

tyttk = parse typedtwotwok

typedtwotwotwok= "("++show (church 2)++")("++show (church 2)++
  ":&C.(&B.(&A.A->B A)->(&A.A->C B A))->"++
  "(&B.(&A.A->(B A))->(&A.A->(C (C B) A)))"++
```

```
  ")("++show (church 2)++")("++show k++")"

tytttk = parse typedtwotwotwok

churchttk = "(\\f:&B^*=>*.(&A^*.A^*->B A^*)->&A^*.A^*->B (B A^*)."++
            "\\x:&A^*.A^*->B^*=>* A^*.((f B^*=>*) x))" {- ++
            "(\\f:A^*->B^*=>* A^*.\\x:A^*.f (f x)"++
            ":&B^*=>*.(&A^*.A^*->B A^*)->&A^*.A^*->B (B A^*)) \\x.\\y.x" -}

typed22k = parse churchttk

-- test my betareduction "be id!"
beid = betaReduction skk

-- type tests

typeds = parse ("(\\x:X^*.(\\y:Y^*.(\\z:Z^*."
                     ++"((x:X^*) (z:Z^*))"
                     ++"(y z))))"
                     ++":((X^*->Y^*)^*->Z^*)^*")
typedk = parse "(\\x:X^*.\\y:Y^*.x):(X->Y->X)^*"
typedi = parse "\\x:X^*.(x:X^*):(X^*->X^*)^*"
typedx = parse "x:X^*"

-- crap

-- t = ((i `App` k) ::: Nothing)
kk = (k `App` k) ::: Nothing
app = parse("x y")
x = parse("x")
y = parse("y")
abstr = parse("\\x.y")
cti = parse("/\\A^*.\\x:A.x") -- kinding of Lambda vars is not sufficient

ts="\\x:Z^*->Y^*->X^*.\\y:Z^*->Y^*.\\z:Z^*.(x z)(y z)"
tk="\\x:Xk^*.\\y:Yk^*.x"
tskk=parse("("++ts++")"++" ("++tk++") "++tk)

tsk=parse("(/\\Z^*./\\Y^*./\\X^*."++ts++
          ") Xk^* Yk^* Xk^*"++" ("++tk++")") -- ++tk)

-- check annotation in abstraction expressions

aexp = parse "\\x:A^*->A^*.x"
```

```
-- illegal generalisation

wrongfa = parse "/\\ A^*. (x:A^*)"

-- Matthes

moniso = "(\\F.&Am.&A.(Am->A) -> F Am -> F A)"
monisoan = "(\\F^*=>*.&Am^*.&A^*.(Am->A) -> F Am -> F A)"

monanti = "(\\F.&Am.&A.(Am->A) -> F A -> F Am)"
monantian = "(\\F^*=>*.&Am^*.&A^*.(Am->A) -> F A -> F Am)"

--mi = parse ("x:"++moniso)
--ma = parse ("x:"++monanti)

-- 1.2

prmi = moniso++"\\A.B"
prmian = monisoan++"\\A^*.B^*"

prma = monanti++"\\A.B"
prmaan = monantian++"\\A^*.B^*"

proofproj = parse "/\\ B^*. /\\ Am^*. /\\ A^*. \\ f:Am^*->A^*. \\ x:B^*. x"

-- 1.3
idmi = moniso++"\\A.A"
idmian = monisoan++"\\A^*.A"

proofidmi = parse ("/\\Am./\\A.\\f:Am->A.\\x:Am^*.f x"++":"++idmi)
proofidmian = parse ("/\\Am^*./\\A^*.\\f:(Am->A)^*.\\x:Am^*.f x"++
                     ":"++idmian)
proofidmian2 = parse ("/\\Am^*./\\A^*.\\f:(Am^*->A^*)^*.\\x:Am^*.f x"++
                     ":"++idmian)

-- 1.4
{-
proofsumpresisotone =  parse $ "/\\F./\\G."++
                     "\\x:"++monisoan++"F."++
                     "\\y:"++monisoan++"G."++
                     "/\\Am./\\A."++
                     "\\f:Am->A."++
                     "\\z:F Am % G Am."++
```

```
                      "z{x1:F Am.INL(x Am A f x1): G A + F A, "++
                      "  y1:G Am.INL(y Am A f y1): G A + F A}"++
                      ":&F.&G.("++monisoan++"F)->"++
                      "("++monisoan++"G)->(\\A.F A + G A)"

proofprodpresisotone = parse $ "/\\F^*=>*./\\G^*=>*."++
                      "\\x:"++monisoan++"F^*=>*."++
                      "\\y:"++monisoan++"G^*=>*."++
                      "/\\Am^*./\\A^*."++
                      "\\f:Am^*->A^*."++
                      "\\z:F^*=>* Am^* % G^*=>* Am^*."++
                      "<x, y>"++ -- "<x Am^* A^* f (z L), y Am^* A^* f (z R)>"++
                      ":&F^*=>*.&G^*=>*."++monisoan++"F^*=>*)->"++
                      "("++monisoan++"G^*=>*)->(\\A^*.F A % G A)"
proofprodpresisotone2 = parse $
                      "/\\F^*./\\G^*."++
                      "\\x:"++monisoan++"F^*=>*.x " --  ++
                      "\\y:"++monisoan++"G."++
                      "/\\Am./\\A."++
                      "\\f:Am->A."++
                      "\\z:F Am % G Am."++
                      "<x Am A f (z L), y Am A f (z R)>"  ++
                      ":&F.&G.("++monisoan++"F)->"++
                      "("++monisoan++"G)->(\\A.F A % G A)"
-}

-- 1.5

-- 1.6

-- 1.7

-- 1.8

-- mail

proofpair = parse "\\ x:A^*.<x,x>"
prooffapair = parse "/\\ A^*. \\ x:A^*.<x,x>"

proofsum = parse "\\ x:A^*.(INL x:A+B^*)"
prooffasum = parse "/\\ A^*. \\ x:A^*.(INL x:A+B^*)"

proofsumiso = parse $ "(/\\ A^*. /\\ B^*. \\ f:A^*->B^*. \\ c:A^*+A^*. "++
                      "(INL (c{cl:A^*.f cl:B^*, cr:A^*.f cr:B^*}):B^*+B^*)"++
```

```
                    "):("++monisoan++")\\X^*.X + X"
proofprodiso = parse $
  "/\\ A^*. /\\ B^*. \\ f:A^*->B^*. \\ p:A^*%A^*. <f(p L),f(p R)>"++
  ":("++monisoan++")\\X^*.X % X"
```

3.1.6 Das Makefile

```
GHC=ghc
HUGS=hugs
HAPPY=happy
RM=rm
PERL=perl
MAKE=make

DEBUG=
GHC_OPTS=-package lang -package net

TEXS=KS.tex HistorySKK.tex ChurchNumerals.tex UrzyChains.tex \
     MonotonicityWitnesses.tex Tabular.tex SimplePair.tex \
     ProofProjIsIsotone.tex ProofSumIsIsotone.tex \
     ProofPairingIsIsotone.tex ProofIdentityIsIsotone.tex \
     WrongForall.tex TypedSK.tex Proof22KIsTypable.tex \
     ProofProdPresIsotone.tex ProofSumPresIsotone.tex

MAINS=Parser CPFwCGI ParseCPFw

.PHONY: all
all: ${TEXS} ${MAINS}

.PHONY: ${MAINS} ${basename ${TEXS}}
${MAINS} ${basename ${TEXS}}: %: %.hs
EXIT=(1 1); \
until test "$${EXIT[1]}" == "0"; do \
  ${GHC} ${GHC_OPTS} --make $< -o $@ 2>&1 \
  | ${PERL} -p \
            -e 'BEGIN {$$remade=0;}' \
            -e "if (/(^ghc.*\: can't find module )\`(.*)'.*$$/ || " \
            -e '    /^(Skipping  |Compiling ).*\( (.*)\.hs\, .* \)$$/ )' \
                    -e '  {if (system "${MAKE} -q $$2.hs")' \
                    -e '    {if (!system "${MAKE} $$2.hs") {$$remade=1}}}' \
            -e 'END {exit $$remade;}'; \
  EXIT=($${PIPESTATUS[*]}); \
done; \
exit $${EXIT[0]}
```

```
%.hs: %.y
${HAPPY} -a ${DEBUG} $< -o $@

%.info: %.y
${HAPPY} -a -i $<

${TEXS}: %.tex: %
./$< > $@

.PHONY: tar
tar:
cvs checkout -d /tmp/fopra/ fopra
WD=$$(pwd); cd /tmp/; tar -cvjf $$WD/fopra.tar.bz2 fopra/
rm -fr /tmp/fopra

.PHONY: GHCVersion.tex
GHCVersion.tex:
( echo "{\tt " ; ${GHC} --version ; echo " }" ) > $@

.PHONY: HugsVersion.tex
HugsVersion.tex:
( echo "{\tt " ; \
  (echo ":version" | ${HUGS} 2>/dev/null | perl -n \
       -e 'if (/^Prelude> -- (Hugs Version .+)$$/) {print $$1 }'); \
  echo " }" ) > $@

.PHONY: clean
clean:
${RM} -f *.o
${RM} -f *.hi
${RM} -f ParseCPFw.hs
${RM} -f ParseCPFw.info
${RM} -f ${TEXS}
${RM} -f ${basename ${TEXS}}
${RM} -f ${MAINS}
${RM} -f GHCVersion.tex
${RM} -f HugsVersion.tex
${RM} -f fopra.tar.bz2
```

Literaturverzeichnis

[1] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[2] Free on-line dictionary of computing. http://www.foldoc.org/, 1994.

[3] Fritz Henglein. *Polymorphic Type Inference and Semi-Unification.* PhD dissertation, Graduate School - New Brunswick, Rutgers, April 1989.

[4] Yan Jurski Hubert Comon. Higher-order matching and tree automata. In *Proceedings of CSL 1997*, number 1414 in Lecture Notes in Computer Science, pages 157–176. Springer Verlag, 1997.

[5] G. P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[6] Mark P. Jones. Typing haskell in haskell. http://www.cse.ogi.edu/ mpj/thih/, 1999. Haskell Workshop Version: Sep 1, 1999.

[7] Ralph Matthes. Datatypes for complex term trees. unpublished, June 2001.

[8] Ralph Matthes. Monotone inductive and coinductive constructors of rank 2. In Laurent Fribourg, editor, *Proceedings of CSL 2001*, number 2142 in Lecture Notes in Computer Science, pages 600–614. Springer Verlag, 2001.

[9] Ralph Matthes. Monotonicity proofs for the csl01 paper. unpublished, January 2002.

[10] Simona Ronchi Della Rocca Paola Giannini, Furio Honsell. Type inference: Some results, some problems. *Fundamenta Informaticae*, 19:87–125, 1993.

[11] Frank Pfennig Rowan Davies. Intersection types and computational effects. ICFP2000, 2000. ???

[12] Pawel Urzyczyn. Type reconstruction in f_ω. *Mathematical Structures in Computer Science*, 7:329–358, 1997.