System F im Lambda-Cube

Seminar: Perlen der Informatik 3 bei Prof. Nipkow

Daniel Stüwe

4. November 2014

Inhaltsverzeichnis

1 Einleitung

Die Einführung der Typisierung im $\lambda_\rightarrow$ zielte darauf ab, Ausdrücke zu verhindern, die nicht zu einem Wert reduziert werden können. Leider lassen sich mit den dort getroffenen, einfachen Regelungen keine allgemeinen Funktionen mehr programmieren; die Typisierung beschneidet also die Ausdrucksstärke des Kalküls stark. In dieser Seminar-Arbeit werden daher Erweiterungen des Typ-Systems vorgestellt, sodass die neu errungene Eigenschaft, dass sich alle Terme zu Werten evaluieren lassen, erhalten bleibt, der Umfang an darstellbaren Funktionen hingegen wesentlich steigt.[1]

Grenzen des einfach getypten Lambda-Kalküls $\lambda_\rightarrow$

Im Folgenden eine beispielhafte Auflistung sinnvoller Funktionen, die sich zwar im ungetypten Lambda-Kalkül, nicht aber im einfach getypten wiedergeben lassen:

- id = λx. x
- $\circ$ = λg. λf. λx. g (f x)
- map, cons, nil etc.

Diese Terme haben kein allgemeines Äquivalent im $\lambda_\rightarrow$, da die Eingabe-Parameter von einem vorher bestimmtem Typ sein müssen. Daher müsste jeweils eine eigene Funktion geschrieben werden je Typ, also id_{Nat} = λx:Nat. x, $\mathrm{id}_{Boolean}$ = λx:Boolean. x usw. Die entscheidende Gemeinsamkeit dieser Funktionen dabei jedoch ist, dass der abstrahierte Term gleich ist und die Abstraktionen sich lediglich in der Typannotation unterscheiden.

Idee

Die Lösung liegt also nahe: Abstraktion über Typen. Anstatt Typen vorher festzulegen werden sie erst zur Laufzeit übergeben. Es werden dazu Typvariablen eingeführt. Für die obigen Beispiele sähe das Ganze also wie folgt aus:

- id = λT. λx:T. x
- $\circ$ = λA. λB. λC. λg:B$\rightarrow$C. λf:A$\rightarrow$B. λx:A. g (f x)
- map[Nat], cons[Float], nil[Float $\rightarrow$ Float] etc.

Diese Art der Polymorphie hat mittlerweile weite Verbreitung gefunden und existiert in allen relevanten statistisch typisierten, funktionalen Sprachen, firmiert bei Java unter dem Namen Generics und die C++-Schöpfer haben ihre Variation Templates getauft.

[1] Diese Arbeit basiert auf [Pie02, Kapitel 22, 23, 29 und 30]

2 Definition von System F

Unter Typ-Theoretikern ist dieser Kalkül als System F, Lambda-Kalkül zweiter Ordnung oder polymorpher Lambda-Kalkül nach Girard–Reynolds bekannt. Jean-Yves Girard (1947*, Lyon) und John C. Reynolds (1. Juni 1935*, 28. April 2013†) haben unabhängig voneinander den Kalkül entwickelt und auf seine Eigenschaften hin untersucht.

2.1 Syntax

Die Syntax, also die Schreibweise ohne semantische Bedeutung, wird durch die untere Grammatik in Backus-Naur-Form festgelegt. Insbesondere werden die später verwendeten Ausdrücke durch die Teilmenge der Terme dargestellt. Die darüberhinausgehenden Definitionen von Wert und Kontext dienen allein der Regelung von Evaluation und Typisierung. Insbesondere stellen Typen außerhalb von Termen keinen Ausdruck des Kalküls dar.

Terme	t ::=	x	$\mid$	λx:T. t	$\mid$	t t	$\mid$	λX. t $\mid$ t [T]
Werte	v ::=	λx:T. t	$\mid$	λX. t				
Typen	T ::=	X	$\mid$	T $\to$ T	$\mid$	$\forall$X. T		
Kontext	Γ ::=	$\emptyset$	$\mid$	Γ, x:T	$\mid$	Γ, X		

2.2 Evaluation

Genau wie die Syntax ist die Evaluation des System eine Evolution des $\lambda_{\to}$. Es wurden die Regeln E-TApp und E-TAppAbs hinzugefügt, um typabstrahierte Terme auswerten zu können.

$$\frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2} \quad E-App_1, \qquad (\lambda X.t_{12})[T_2] \to [X \mapsto T_2]t_{12} \quad E-TAppAbs$$

$$\frac{t_1 \to t_1'}{t_1[T_2] \to t_1'[T_2]} \quad E-TApp, \qquad (\lambda x : T_{11}.t_{12})v_2 \to [x \mapsto v_2]t_{12} \quad E-AppAbs,$$

$$\frac{t_2 \to t_2'}{v_1 t_2 \to v_1 t_2'} \quad E-App_2,$$

2.3 Typisierung

Die wichtigen Neuerungen dieses Kalküls betreffen natürlich das Typ-System. Wieder gilt es, besondere Aufmerksamkeit der Typ-Abstraktion mit den Regeln

T-TAbs und T-TApp zu widmen. Typabstraktion kann durch Applikation aufgelöst werden, der angegebene Typparameter wird dabei in den Typ substituiert.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad T - Var \qquad \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash (\lambda x : T_1 . t_2) : T_1 \to T_2} \quad T - Abs$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X . t_2 : \forall X . T_2} \quad T - TAbs \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2} \quad T - App$$

$$\frac{\Gamma \vdash t_1 : \forall X . T_1}{\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_1} \quad T - TApp$$

3 Eigenschaften

Die nachfolgenden Theoreme bleiben hier ohne Beweis, da sie, vor allem die letzteren, alles andere als trivial sind und somit den Rahmen dieser Arbeit bei weitem überschreiten. Nichtsdestotrotz sind die gewonnen Erkenntnisse bedeutsam, angesichts der Konstruktionsziele des Kalküls und seiner daraus entspringenden Ausdrucksstärke.

Satz (Preservation)**.** *Falls* $\Gamma \vdash t : T$ *und* $t \longrightarrow t'$*, dann* $\Gamma \vdash t' : T$*. (Unter Evaluation bleibt der Typ erhalten.)*

Satz (Progress)**.** *Falls t ein wohlgetypter, geschlossener Term ist, dann ist t ein Wert oder es gibt ein t', sodass* $t \longrightarrow t'$*. (Alle Nicht-Werte lassen sich evaluieren.)*

Satz (Normalization - Girard, 1972)**.** *Falls t ein wohlgetypter Term ist, so gibt es einen Wert v, sodass* $t \longrightarrow^* v$*.*

Es ist sehr erstaunlich, dass dieser Satz gilt, da er bedeutet, dass jedes „Programm“ geschrieben im System F terminiert.

Aus dem Preservation-Theorem, also der Invarianz des Typs eines Terms unter Evaluation folgt:

Satz (Type Erasure)**.** *Es ist gibt eine Funktion* $erase(t)$*, die Terme des System F auf jene des ungetypten Lambda-Kalküls abbildet, ohne dass dabei die Evaluation der jeweiligen Terme geändert wird.* ($erase(t)$ *ist also ein Homomorphismus zwischen den Kalkülen bezüglich der Evaluation.)*

Die Umkehrung gilt bedauerlicherweise nicht. Typrekonstruktion ist somit im Allgemeinen nicht möglich:

Satz (Typrekonstruktion - Wells, 1994)**.** *Es ist unentscheidbar, ob zu einem Term t des ungetypten Lambda-Kalkül es einen Term* t' *gibt im System F, sodass* $erase(t') = t$*.*

4 Beispiele

In diesem Abschnitt wenden wir uns der Programmierung im System F zu. Wir interessieren uns im Speziellen für genau die Terme, welche sich im $\lambda_{\rightarrow}$ nicht typisieren lassen.

4.1 Funktionale

Dies trifft u. a. auf Funktionen höherer Ordnung zu. Also Funktionen, die wiederum Funktionen als Eingabe übergeben bekommen. Diese lassen sich nicht zufriedenstellend im $\lambda_{\rightarrow}$ realisieren. Das System F hingegen bietet auf intuitive Weise die benötigten Werkzeuge an. Betrachten wir dazu die double-Funktion, die eine Funktion f zweimal auf die Eingabe anwendet. [2]

```
double : ∀X. (X → X) → X → X
double = λT. λf:T→T. λx:T. f (f x)
```

Diese Funktion lässt sich sogar auf sich selbst anwenden, wenn sie entsprechend abgewandelte Typparameter erhält!

```
quad   : ∀X. (X → X) → X → X
quad   = λT. double[T→T] double[T]
```

Dabei evaluiert in zwei Schritten quad $\rightsquigarrow^*$ quad'.[3] Der Typ bleibt nach dem Preservation-Theorem dabei natürlich der selbe.

```
quad'  = λT. λx:T→T. double[T] (double[T] x)
```

Solche Terme waren im $\lambda_{\rightarrow}$ ausgeschlossen, da dort ein Typ entweder von der Form $T \rightarrow T$ oder T ist. Im System F ist es nun möglich, dass ein typallquantifizierter Parameter beide Formen annehmen kann.

4.2 Church Encodings

Im Lambda-Kalkül lassen sich Ausdrücke finden, die isomorph zu bekannten algebraischen Strukturen sind. Alonzo Churchs Kodierung bool'scher Werte wollen wir im System F nun typisieren. Ungetypt bilden dabei tru' und fls' die Grundsteine der Kodierung.

```
tru' = λt. λf. t          fls' = λt. λf. f
```

Wir verwenden zunächst informell $\mathrm{CBool} = \forall X.\ X \rightarrow X \rightarrow X$ als abkürzende Schreibweise für ebenjenen Typ von tru und fls, welche die getypten Terme zu tru' beziehungsweise fls' sind:

[2] Die Typ-Notation mittels Doppelpunkt ist kein Ausdruck im System F. Sie soll ausschließlich dem Leser behilflich sein.

[3] Genau genommen, wird unter Abstraktion nach gegebenen Inferenzregeln keine Evaluation durchgeführt, sodass man eigentlich eher von einer Konversation zwischen quad und quad' sprechen müsste. Vergleiche β-Reduktion bzw. -Konversation im Lambda-Kalkül.

```
tru  = λX. λt:X. λf:X. t      fls  = λX. λt:X. λf:X. f
```

An diesem Beispiel lässt sich leicht nachvollziehen, dass durch Entfernen der Typannotation und -abstraktionen sich der äquivalente Term im ungetypten Lambda-Kalkül erzeugen lässt. Außerdem sei hier die Konvention nachträglich erwähnt, dass Typen implizit rechtsassoziativ geklammert werden.

```
not : CBool → CBool
not = λb:CBool (λX. λt:X. λf:X. b[X] f t)

and : CBool → CBool → CBool
and = λa:CBool λb:CBool
            λX. λt:X. λf:X. a[X] (b[X] t f) f
```

4.3 Listen

Ein fortgeschrittenes Beispiel aus dem ungetyptem Lambda-Kalkül sind Listen. Wir nutzen auch hier wieder die informelle Kurzform des Types:

```
List X = ∀R. (X → R → R) → R → R

nil  : ∀X. List X
nil  = λX. λR. λc:X→R→R. λn:R. n

cons : ∀X. X → List X → List X
cons = λX. λhd:X. λtl. List X.
             λR. λc:X→R→R. λn:R. c hd (tl[R] c n)
```

Die grundlegende Vorgehensweise ähnelt der, der Church-Numerale. Listen werden durch eine Funktion repräsentiert, die in funktionalen Programmiersprachen foldr oder reduce genannt wird. Listen lassen sich verarbeiten, indem man ihnen eine Faltungsfunktion und Basiselement übergibt. So ist es möglich, Listen zu sortieren! Hier nach dem Insert-Sort-Algorithmus.[4]

```
insert : ∀X. (X→X→Bool) → List X

sort   : List X → List X
sort   = λX. λleq: X→X→Bool. λxs:List X.
           xs[List X]
           (λhd:X. λrest:List X. insert[X] leq rest hd)
           (nil[X])
```

Der entscheidende Trick dabei ist, dass man mit einer leeren Akkumulator-Liste als Basiselement beginnt und dann mit insert die Liste zusammenfaltet.

[4] Siehe [Pie02, S. 547 f.]

5 Fragmente des System F

5.1 Motivation

Zwar gewinnen wir durch die Typabstraktion ein mächtiges Instrument um generische Funktionen zu schreiben, verlieren jedoch die angenehme Eigenschaft, Typen rekonstruieren zu können. Im Hinblick auf reale Programmiersprachen ist dies sehr bedauerlich, da Entwickler lieber auf die, ihrer Meinung nach häufig redundanten, Typ-Signaturen verzichten wollen.

5.2 Let-Polymorphismus

Als nützliche Teilmenge des System F zeigten sich die Ausdrücke, die sich nach einem Verfahren vom Hindley und Milner einen Typ zuweisen lassen. Im Wesentlichen darf dabei ein Typ nicht mehr polymorph sein, wenn er gewöhnlich abstrahiert ist.

```
λf. (f true, f 5)
```

Der obige Term lässt sich nicht typisieren, da also f von der Form Bool → T und Nat → T sein müsste, dies geht jedoch nicht.[5] Nachfolgendes jedoch lässt sich einen Typ zu weisen:

```
let f = λx.x in (f true, f 5)
```

Das System von Hindley und Milner erkennt in diesem Fall, da f bereits konkret bekannt ist, den Term als typkorrekt an. Der von ihnen entwickelte Algorithmus terminiert in der Praxis schnell, hat jedoch eine exponentielle Worst-Case-Laufzeit. Die ML-Sprachfamilie, darunter auch Haskell nutzen dieses Verfahren zur Typinferenz.

5.3 Rang-2-Polymorphismus

Das System F lässt sich auch als Vereinigung aller Terme aller Ränge darstellen. Diese Beispiele sollen dabei intuitiv den Begriff Rang erklären:[6]

0. Nat → Nat, (Nat → Nat) → Nat → Nat, Bool
1. ∀X. X → Bool, ∀X. ∀Y. (X → Y) → X → Y
2. Nat → (∀X. X → Bool) → Bool
3. Nat → (Nat → (∀X. X → X) → Nat) → Nat

[5] In Systemen mit Subtyping und einem gemeinsamen Obertyp TOP hingegen schon.
[6] Die Enumeration gibt den Rang der Typen an.

Es fällt auf, dass alle bisherigen (und alle nachfolgenden) Beispiele höchstens vom zweitem Rang waren. Ende der 1980iger Jahre konnte für diese bedeutende Teilmenge des System F gezeigt werden, dass für jene Terme Typrekonstruktion entscheidbar ist und entsprechende effiziente Algorithmen wurden anschließend gefunden. Die oben aufgeführte Funktion λf. (f true, f 5) ist vom zweitem Rang und könnte in diesem System also getypt werden.

6 Ausblick

6.1 Skizze des einfach getypten Lambda-Kalküls mit Typoperatoren λ_ω

6.1.1 Motivation

In den oberen Abschnitten wurde teilweise die Schreibweise List X, Pair X Y etc. verwendet. Ziel ist es nun, dies zu formalisieren. Dazu beginnen wir erst mit dem einfach getypten Lambda-Kalkül $\lambda_\rightarrow$ und erweitern zunächst nur dieses.
Betrachten wir vorerst ein Beispiel:

```
Pair      = λY. λZ. ∀X. (Y → Z → X) → X

Pair S T = (λY. λZ. ∀X. (Y → Z → X) → X) S T
         = (S → T → X) → X
```

Man verwendet also auch hier wieder Typvariablen, die Applikation findet dabei allerdings auf Ebene der Typen statt! In diesem System ist es jedoch nicht zulässig, Terme mit allquantifizierten Typen zu versehen. Alle Typvariablen müssen vorher durch Anwendung entfernt worden sein.

6.1.2 Kinds

Wie im ungetypten Lambda-Kalkül können nun auch hier wieder sinnlose Ausdrücke erstellt werden, jedoch auf Ebene der Typen, nicht der Terme. Z. B. Bool Nat, also die Anwendung von Bool auf Nat. Um das zu verhindern führt man im λ_ω sogenannte Kinds ein - Typen der Typen. Dabei wird hauptsächlich zwischen Arrow- und Basis-Typen unterschieden, um unzulässige Applikation wie im Beispiel zu verhindern. Arrow-Typen sind dabei Funktionen, die Typen nehmen und einen Typ zurückgeben.[7]

[7] Im Anhang gibt es eine Graphik, die dem Buch entnommen ist, die die Relation zwischen Termen, Typen und Kinds gut darstellt, nur leider etwas unscharf ist.

6.2 System F_ω

Verbindet man diese Idee mit dem System F, ergibt dies einen Kalkül, System F_ω genannt. So wird die Eleganz in der Notation der Typen im Vergleich zum puren System F merklich gesteigert. Jedoch lassen sich keine neuen Terme des ungetypten Lambda-Kalküls typisieren. Die Einführung der Kinds kann aber in Abwandlungen dazu genutzt werden, um darüber hinaus Typen noch genauer zu unterscheiden. Etwa in Haskell, dort differenziert man zwischen boxed und unboxed Datentypen.[8]

6.2.1 Definition des System F_ω

Syntax

Es wird syntaktisch allen Typvariablen ein Kind zugeordnet. Und Kinds werden in die Sprache aufgenommen.

Terme t	::=	x | λx:T. t | t t | λX::K. t | t [T]
Werte v	::=	λx:T. t | λX::K. t
Typen T	::=	X | T → T | ∀X::K. T | λX::K. T | T T
Kontext Γ	::=	∅ | Γ, x:T | Γ, X::K
Kind K	::=	* | K ⇒ K

Evaluation

An der Auswertungsstrategie des Kalküls ändert sich semantisch nichts, wir müssen lediglich die Kind-Anmerkung syntaktisch aufnehmen in E-TAppAbs.

$$\frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2} \quad E-App_1, \qquad (\lambda X :: K.t_{12})[T_2] \to [X \mapsto T_2]t_{12} \quad E-TAppAbs$$

$$\frac{t_1 \to t_1'}{t_1[T_2] \to t_1'[T_2]} \quad E-TApp, \qquad (\lambda x : T_{11}.t_{12})v_2 \to [x \mapsto v_2]t_{12} \quad E-AppAbs,$$

$$\frac{t_2 \to t_2'}{v_1 t_2 \to v_1 t_2'} \quad E-App_2,$$

[8] Unboxed Typen sind immer strikt in der Auswertung und haben ein entsprechendes C-Pendant, boxed Typen sind gewöhnliche Haskell-Typen. [has14]

Kinding

Neu eingeführt sind die Kinding-Regeln. Sie sind denen der Typisierung im $\lambda_{\rightarrow}$ sehr ähnlich, jedoch etwas vereinfacht.

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \qquad \frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1.T_2 :: K_1 \Rightarrow K_2}$$

$$\frac{\Gamma \vdash T_1 :: K_1 \Rightarrow K_2 \quad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1 T_2 :: K_2} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X :: K_1.T_2 :: *}$$

Typ-Äquivalenz

Es muss eine Äquivalenz-Relation zwischen den Typen angegeben werden, da es nun möglich ist, das syntaktisch unterschiedliche Typen den semantisch selben darstellen. Die Regeln sind hier der Vollständigkeit halber aufgelistet.

$$T \equiv T \qquad \frac{S \equiv T}{T \equiv S} \qquad \frac{S \equiv T \quad T \equiv U}{S \equiv U}$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2} \qquad \frac{S \equiv T}{\forall X :: K_1.S \equiv \forall X :: K_1.T} \qquad \frac{S \equiv T}{\lambda X :: K_1.S \equiv \lambda X :: K_1.T}$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 S_2 \equiv T_1 T_2} \qquad (\lambda X :: K.T_2)T_3 \equiv [X \mapsto T_3]T_2$$

Typisierung

Interessant ist hier die Regel T-Eq, die die Typ-Äquivalenz einbezieht. Aber auch auf die Regel T-TApp ist hinzuweisen, da dort der angewendete Typ T_2 vom gleichen Kind sein muss, wie die Typvariable selbst um eben gerade unsinnige Typ-Applikation zu verbieten.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ T - Var \qquad \frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}\ T - App$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash (\lambda x : T_1.t_2) : T_1 \rightarrow T_2}\ T - Abs \qquad \frac{\Gamma, X :: K \vdash t_2 : T_2}{\Gamma \vdash \lambda X :: K.t_2 : \forall X.T_2}\ T - TAbs$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_1 :: K \quad \Gamma \vdash T_2 :: K}{\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_1}\ T - TApp \qquad \frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T}\ T - Eq$$

6.2.2 Beispiel - Tupel zu Tripel

Wenn wir unser Beispiel aus λ_ω aufgreifen und eine generische Funktion add schreiben, die ein Tupel beliebigen Subtyps nimmt, einen weiteren beliebigen Wert und ein Tripel zurückgibt, erhalten wir:

```
add : ∀R. ∀S. ∀T. Pair R S → T → Triple R S T

add = λR. λS. λT. λp:Pair R S. λt:T.
        λtrpl: (∀U. ∀V. ∀W. U→V→W→X).
           trpl[R][S][T] (fst[R][S] p) (snd[R][S] p) t
```

6.3 Dependent Types

6.3.1 Motivation

Wir sprachen bereits über Kalküle mit folgenden Elementen:

λx: T_1.t_2	Abstraktion über Terme durch Terme
λX:: K_1.t_2	Abstraktion über Terme durch Typen
λX:: K_1.T_2	Abstraktion über Typen durch Typen

Es ist nun ganz offensichtlich, dass in dieser Auflistung noch die Abstraktion der Typen durch Terme fehlt. Oder anders formuliert, sollen Terme nun Typen parametrisieren, wie ja bereits im System F mit Typen bei Termen gesehen. Dieses Gebiet wird unter dem Begriff Dependent Types zusammengefasst.

6.3.2 Einfach getyptes Lambda-Kalkül mit abhängigen Typen (LF)

Dazu ein Beispiel aus LF. Wie erweitern wieder lediglich $\lambda_\rightarrow$. Also werden hier Listen mit Elementen des konkreten Typs Float benutzen. Wir parametrisieren nun diesen einfachen Typ mit einer Zahl, um die Länge der Liste im Typ zu kodieren.

```
nil   : FloatList 0
cons  : Πn:Nat. Float → FloatList n → FloatList Succ(n)
hd    : Πn:Nat. FloatList Succ(n) → FloatList
cons  : Πn:Nat. FloatList Succ(n) → FloatList n
```

Der Vorteil liegt ganz klar auf der Hand, es wird so durch Typen statisch verhindert, dass wir hd auf eine leere Liste aufrufen. Der Nachteil, den wir dadurch erkaufen, ist, dass Typüberprüfung damit de facto einem Beweis erzeugen muss, der zeigt, dass im Verlauf unseres Programmes die Eingabe von hd vom Typ „Nicht-Leere-Liste“ sein muss. Und im Allgemeinen ist so etwas natürlich nicht entscheidbar.

6.3.3 Calculus of Constructions - CC

Trotzdem lohnt sich ein Blick auf die Zusammenführung dieser Idee mit dem System F_ω. Wir verallgemeinern dazu obiges Beispiel zu allquantifizierten Funktionen, die beliebige Listen annehmen:

```
nil  : ∀X. List X 0
cons : ∀X. Πn:Nat. X → List X n → List X Succ(n)
hd   : ∀X. Πn:Nat. List X Succ(n) → X
cons : ∀X. Πn:Nat. List X Succ(n) → List X n
```

6.4 Barendregts Lambda-Würfel

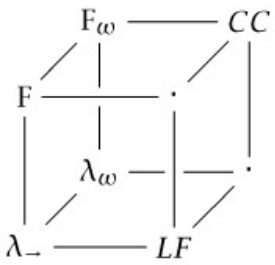

Abbildung 1: Barendregt's Lambda Cube [Pie02, S. 465]

Diese Abbildung zeigt anschaulich die Zusammenhänge der einzelnen Typ-Systeme ausgehend vom einfach getypten Lambda-Kalkül unten links. Dabei entspricht der vertikale Pfeil dem Hinzufügen allquantifizierter Typen, der nach hinten gehende dem der Typoperatoren und schlussendlich der nach rechts gehende Pfeil, dem der zuletzt vorgestellten Abstraktion der Typen durch Terme also der Dependent Types.

7 Anhang

Kinds

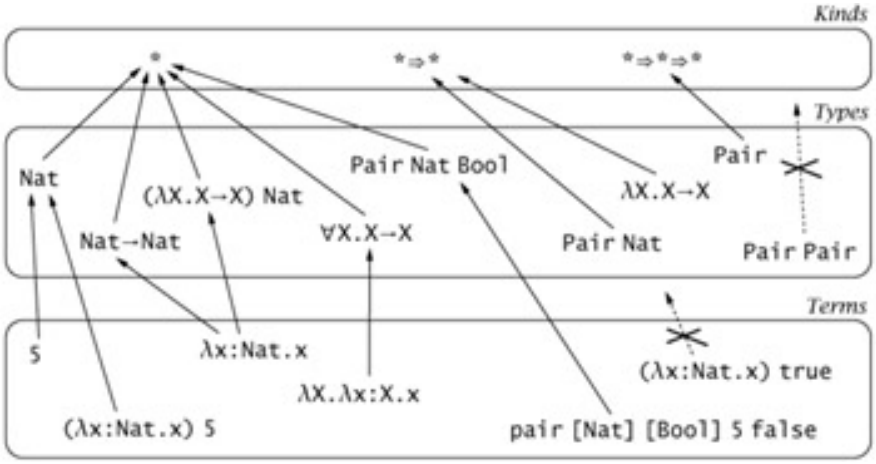

Abbildung 2: Kinding Relation [Pie02, S. 444]

Literatur

[has14] `https://www.haskell.org/haskellwiki/Unboxed_type`, 2014. Stand: 03.11.2014.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, 2002.