Fortgeschrittenenpraktikum
Implementierung eines Type Checkers für das System F_ω

Jan Peter Gutzmann

19.12.03

Inhaltsverzeichnis

Kapitel 1

Einleitung

In (fast) allen Programmiersprachen der heutigen Zeit sind sie allgegenwärtig: Typen [4]. Ob in C oder Java, jeder benutzt ständig und ohne groß darüber nachzudenken Variablen *vom Typ Integer*. Auch Funktionen haben Typen: Eine Funktion $sqr(x)$ beispielsweise, die das Quadrat einer reellen Zahl x berechnet, hat den Typ $Real \rightarrow Real$, d.h. sie nimmt ein Argument x vom Typ *Real* und liefert ein Ergebnis (nämlich x^2) vom Typ *Real*.

Doch wie stellt man sicher, dass die Funktion als Argument wirklich einen *Real* - Wert erhält und niemand beispielsweise versucht, $sqr(\text{„}hallo\text{“})$ aufzurufen? Hierzu muss man den Programmcode einem Prozess unterziehen, in dem für jedes Argument, jeden Parameter und jede Funktion festgestellt wird, ob der angegebene Typ mit dem Typ übereinstimmt, den der Ausdruck eigentlich haben sollte. So würde dieser Prozess bei obigem Beispiel „hallo“ als *String* erkennen und einen Fehler feststellen, da ein *String* kein „Integer“ ist. Diesen Prozess nennt man **Typprüfung** oder auf Englisch **Type Checking**.

1.1 Der Lambda-Kalkül

Alle funktionalen Programmiersprachen basieren auf dem *Lambda-Kalkül*. Er wird in vielen Büchern und Abhandlungen [1] ausgiebig beschrieben und behandelt, daher wird hier nur ein kurzer Überblick gegeben.

1.1.1 Der Ungetypte Lambda-Kalkül

Der Lamda-Kalkül besteht im wesentlichen aus drei Konstrukten:

- Der **Abstraktion**:

  Eine Abstraktion hat folgende Gestalt:

  $$\lambda x.t$$

  Das λx bedeutet, dass diese Abstraktion eine Funktion darstellt, welche ein Argument x erhält. Die Funktion liefert t als Ergebnis. Ein Beispiel:

  $$\lambda x.x$$

  ist die Identitätsfunktion. Sie erhält ein Argument x und gibt dieses als Ergebnis zurück.

  Durch sogenanntes *Currying* lassen sich Funktionen mit mehr als einem Parameter bilden:

  $$\lambda x_1.\lambda x_2....\lambda x_n.t$$

  Dies ist eine Funktion mit n Parametern, welche als Rückgabewert t berechnet. Einige Beispiele:

  Church-Codierung von Paaren:

  $$pair = \lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y$$

  $$fst = \lambda p.\ p\ (\lambda x.\ \lambda y.\ x)$$

  $$snd = \lambda p.\ p\ (\lambda x.\ \lambda y.\ y)$$

- Der **Applikation:**

  Eine Applikation hat folgende Gestalt:

  $$f\ e$$

  Das hat folgende Bedeutung: ist f eine Abstraktion, so ersetze alle Vorkommnisse der Abstraktionsvariablen durch e. Falls f keine Abstraktion ist, bleibt der Ausdruck so stehen. Ein Beispiel:

  $$fst\ (pair\ 2\ 0) \longrightarrow pair\ 2\ 0\ (\lambda x.\ \lambda y.\ x) \longrightarrow (\lambda x.\ \lambda y.\ x)\ 2\ 0 \longrightarrow 2$$

- **Variablen:**

  Man kann in Termen auch jede beliebige Variable benutzen, die nicht durch eine λ-Abstraktion gebunden ist (Bsp: x, y, z). Diese freien Variablen sollten jedoch von ausserhalb mit einem Wert versehen werden (Bsp: $pair = \lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y$)

1.1.2 De Bruijn-Repräsentation

Der Name einer Variablen ist für die Bedeutung eines Terms unerheblich.

$$\lambda x.\ x \quad \text{und} \quad \lambda y.\ y$$

sind ein und derselbe Term (bis auf die Namen der Variablen). Aus diesem Grund hat *Nicolas de Bruijn* eine Term-Repräsentation eingeführt[2] (die nun unter seinem Namen als **de Bruijn-Repräsentation** bekannt ist), bei der Variablen nicht durch Namen gekennzeichnet sind, sondern durch den Abstand zu ihrem λ-Binder repräsentiert werden, also durch die Anzahl der λ's, die eine Variable überspringen muss, um zu „ihrem“ Binder zu gelangen.

$$\lambda x.\ x \longrightarrow \lambda.\ 0$$

$$\lambda y.\ y \longrightarrow \lambda.\ 0$$

$$\lambda x \lambda y \lambda z.\ z\ x\ y \longrightarrow \lambda\lambda\lambda.\ 0\ 2\ 1$$

Man beachte, dass die Terme $\lambda x.x$ und $\lambda y.y$ nun beide durch $\lambda.0$ repräsentiert werden. Freie Variablen muss man hier in einem Kontext Γ angeben. Dabei zählt jeder Eintrag in einem Kontext zur Bestimmung des Wertes einer Variablen wie ein λ-Binder. Auch hierzu ein Beispiel:

$$x, y \vdash \lambda z.z\ x\ y \longrightarrow x, y \vdash \lambda.0\ 2\ 1$$

1.1.3 Der Getypte Lambda-Kalkül

Nun kann man auch Typen in den Lambda-Kalkül einführen. Man annotiert eine Variable mit einem Typen, indem man ihn beim Binden angibt:

$$\lambda x : Int.\ x$$

Diese Funktion hat den Typen $Int \rightarrow Int$, d.h. sie nimmt einen Parameter vom Typ Int und liefert einen Ergebniswert vom Typ Int. Auch freie Variablen können annotiert werden. Dies geschieht, indem man im Kontext den Typen mit angibt.

Man unterscheidet zwischen zwei Varianten des getypten Lambda-Kalküls:

- **Church-Style:** Vom Church-Style Lambda-Kalkül spricht man, wenn alle Terme vollständig annotiert sein *müssen*, d.h. für jede Variable muss der Typ angegeben sein (wie z.B. in Java, wo für jede Variable explizit der Typ angegeben sein muss).
- **Curry-Style:** Vom Curry-Style spricht man dann, wenn die Variablen nicht annotiert sind.

Stark-getypte funktionale Programmiersprachen (wie z.B. Haskell oder die ML-Dialekte) benutzen gewöhnlich eine Mischung beider Stile. Variablen können Typannotationen tragen; fehlen diese, versucht der Compiler, die Typen der Variablen aus dem Zusammenhang zu inferieren.

1.2 Ziele des Fortgeschrittenenpraktikums

Ziel dieser Arbeit war es, einen Type Checker für den vollständig annotierten (Church-Style) Lambda Kalkül mit Polymorphismus höherer Ordnung (System F_ω) zu implementieren. Für die interne Repräsentation der Terme wurde eine Darstellung in de Bruijn-Notation gewählt.

Kapitel 2

Das System $\mathbf{F}_\omega$

2.1 Motivation

Betrachtet man noch einmal das Beispiel $\lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y$, also die Funktion zur Paarbildung, so kann man kann sehr einfach Paare von z.B. zwei Ganzzahlen bilden, auf denen Funtionen operieren können, die eine Ganzzahl als Ergebniswert haben:

$$\lambda x\!:\!Int.\ \lambda y\!:\!Int.\ \lambda z\!:\!Int \to Int \to Int.\ z\ x\ y$$

Möchte man Paare zweier reller Zahlen bilden, bei denen z als Ergebnis eine reelle Zahl liefert, so definiert man folgende Funktion:

$$\lambda x\!:\!Real.\ \lambda y\!:\!Real.\ \lambda z\!:\!Real \to Real \to Real.\ z\ x\ y$$

Auf diese Weise ist Paarbildung für je zwei Typen und einen Ergebnistyp von z möglich. Doch es gibt eine Möglichkeit, mit nur einer Deklaration alle möglichen Fälle von Paarbildung abzudecken, und zwar durch **Polymorphismus**.

2.2 Typ-Abstraktion und Typ-Applikation

Hierzu wird zunächst eine **Typ-Abstraktion** benötigt:

$$\Lambda X.\lambda x\!:\!X.\ x$$

Diese Λ-Abstraktion realisiert den gewünschten Polymorphismus: gegeben ein konkreter Typ X, erhält man eine Funktion vom Typ $X \to X$. Der Typ eines

solchen polymorphen Terms ist, da er für jedes beliebige X gilt:

$$\forall X.\ X \to X$$

Auch bei Typ-Abstraktionen kann man durch *currying* Funktionen mit mehr als einem Parameter erzeugen.

$$\Lambda X \Lambda Y.\ \lambda x{:}X \lambda y{:}Y.\ \Lambda Z.\ \lambda z{:}X \to Y \to Z.\ z\ x\ y$$

Hieraus erhält man Funktionen zur Bildung von Paaren bestimmter Typen durch Applikation konkreter Typen auf die Typ-Abstraktion.

Typ-Applikation:

$$e\ [F]$$

Ist e eine Typ-Abstraktion, werden alle Vorkommen der Abstraktionsvariablen durch den konkreten Typ F ersetzt. Hierzu wieder ein Beispiel:

$$(\Lambda X \Lambda Y.\ \lambda x{:}X \lambda y{:}Y.\ \Lambda Z.\ \lambda z{:}X \to Y \to Z.\ z\ x\ y)[Int]\ [Int]$$

gibt eine Funtion zur Bildung von Paaren zweier Ganzzahlen:

$$\lambda x{:}Int \lambda y{:}Int.\ \Lambda Z.\ \lambda z{:}Int \to Int \to Z.\ z\ x\ y$$

Man beachte hier, dass der Typ Z bei der Bildung von Paaren nicht spezifiziert werden muss. Dadurch erhält man die Möglichkeit, auf ein Paar Funktionen mit verschiedenen Rückgabetypen anzuwenden.

$fst\ [Int]\ [Int] = \lambda p{:}\forall Z.\ (Int{-}{>}\ Int{-}{>}\ Z){-}{>}\ Z.\ p\ [Int]\ (\lambda x{:}Int.\ \lambda y{:}Int.\ x)$

2.3 Funktionen auf Typ-Ebene

Abstraktionen und Applikationen werden nun auch auf der Typ-Ebene eingeführt. Die Notation hierfür ist die gleiche wie bei Termen:

$$\lambda X.X$$

ist die Identitätsfunktion für Typen,

$$(\lambda X.X)\ Int$$

liefert wie gewünscht den Typ Int.

2.4 Kinds

Dadurch entsteht die Möglichkeit, „sinnlose“ Typen zu notieren, wie z.B. $Nat\ Bool$. Um dies zu verhindern führt man „Typen für Typen“, sogenannte **Kinds** ein.

Elementare Typen, wie z.B. $Bool$, Nat, $Bool \to Nat$ etc. sind „echte Typen“ und werden mit dem Kind $*$ annotiert. Funktionen auf Typen haben den Kind $K \to K'$. Auch hierzu wieder ein Beispiel:

$$\lambda X : *.\ X$$

ist die Identitätsfunktion auf Typen. Sie nimmt ein Typ-Argument, welches ein echter Typ sein muss, und gibt dieses Argument zurück. Die Identitätsfunktion hat also den Kind $* \to *$.

Damit sinnlose Typen vermieden werden, muss ein Prozess wie das Type Checking für Terme auch für Typen durchgeführt werden. Diesen Prozess nennt man **Kind Checking**. Er ist allerdings ein integraler Bestandteil des Type Checkings und nicht losgelöst davon zu betrachten.

2.5 System $\mathbf{F}_\omega$

Fasst man all das in den vorigen Abschnitten Genannte zusammen, erhält man einen *Lambda-Kalkül mit Polymorphismus höherer Ordnung*, auch **System $\mathbf{F}_\omega$** genannt. Zusammenfassend sieht die Syntax für das System F_ω aus wie folgt:

$t ::=$		Terme:
	x	Variable
	$\lambda x : T.\ t$	Abstraktion
	$t\ t$	Applikation
	$\Lambda X : K.\ t$	Typ-Abstraktion
	$t\ [T]$	Typ-Applikation
$T ::=$		Typen:
	X	Typvariable
	$T \to T$	Funktionstyp
	$\forall X : K.\ T$	Universaltyp
	$\lambda X : K.\ T$	Operator-Abstraktion
	$T\ T$	Operator-Applikation
$K ::=$		Kinds:
	$*$	Kind für Proper Types
	$K \to K$	Kind für Operatoren

Kapitel 3

Details der Implementierung

3.1 Datenstruktur

3.1.1 Interne Repräsentation von Termen

Für die interne Repräsentation von Termen wurde die de Bruijn-Darstellung gewählt. Um Funktionen wie Substitution und Shifting, welche für das Type Checking benötigt werden, nur einmal definieren zu müssen, wurde für Applikation und Abstraktion auf Term- und Typ-Level sowie für Funktionstyp bzw. -kind eine einheitliche Repräsentation gewählt.

Gemeinsame Datenstrukur für Terme, Typen und Kinds

```
data Exp = Var Int
         | Abs String (Maybe Exp) Exp
         | App Exp Exp
               | TyAbs String (Maybe Exp) Exp
               | TyApp Exp Exp
               | Arr Exp Exp
               | Type
               | Forall String Exp Exp
```

Für eine zukünftige Erweiterung zum „Curry-Style“ sind Typannotationen optional; der Typ-Checker kann solche Terme jedoch nicht typisieren.

3.1.2 Kontext

Ein Kontext besteht aus einer Menge von Variablenbindungen. Eine Variablenbindung besteht aus dem Namen der Variablen, sowie einem Eintrag für diese Variable. Ein Eintrag ist entweder eine Definition oder eine durch einen λ-, Λ- oder $\forall$-Binder gebundene Variable. Definitionen bestehen aus einem Wert und einer Typ/Kind-Annotation, gebundene Variablen nur aus einem (optionalen) Typ/Kind. Beide Einträge enthalten ausserdem einen Indikator, ob es sich um eine Term- oder eine Typ-Variable handelt.

Datenstruktur für Kontexte

```
data Indicator = IsType | IsTerm
               deriving Show

data Entry = VarEntry Indicator (Maybe Exp)
           | DefEntry Indicator Exp Exp

type VarBin = (String,Entry)
type Context = [VarBin]
```

3.2 Typ-Prüfung im Sytem $\mathbf{F}_\omega$

3.2.1 Hilfsfunktionen

Lifting. $T \uparrow_c^d$ berechnet den Ausdruck T', in dem alle Variablen mit deBruin-Index $\geq c$ um d erhöht wurden.

$$
\begin{array}{rll}
k \uparrow_c^d & = k + d & \text{if } k \geq c \\
k \uparrow_c^d & = k & \text{else} \\
(\lambda x{:}S.\ T) \uparrow_c^d & = \lambda x{:}(S \uparrow_c^d).\ (T \uparrow_{c+1}^d) & \\
(\Lambda X{:}\kappa.\ T) \uparrow_c^d & = \Lambda X{:}\kappa.\ (T \uparrow_{c+1}^d) & \\
(\forall X{:}\kappa.\ S) \uparrow_c^d & = \forall X{:}\kappa.\ (s \uparrow_{c+1}^d) & \\
(R\ S) \uparrow_c^d & = (R \uparrow_c^d)\ (S \uparrow_c^d) & \\
(T\ [S]) \uparrow_c^d & = T \uparrow_c^d\ [S \uparrow_c^d] & \\
(R \to S) \uparrow_c^d & = (R \uparrow_c^d) \to (S \uparrow_c^d) & \\
* \uparrow_c^d & = * &
\end{array}
$$

Wir führen folgende Abkürzungen ein: $\uparrow$ für $\uparrow_0^1$ und $\downarrow$ für $\uparrow_0^{-1}$

Substitution. Bei der Substitution einer Variablen k durch einen Term/Typ R in einem Term/Typ T werden alle Vorkommen von k ersetzt durch R. Dabei müssen freie Variablen unter einem Binder geliftet werden.

$$
\begin{array}{lll}
[R/k]k & = R & \\
[R/k]l & = l & \text{falls } l \neq k \\
[R/k](\lambda x{:}S.\ T) & = \lambda x{:}([R/k]S).\ ([R\uparrow/k+1]T) & \\
[R/k](\Lambda X{:}\kappa.\ T) & = \Lambda X{:}\kappa.\ ([R\uparrow/k+1]T) & \\
[R/k](\forall X{:}\kappa.\ S) & = \Lambda X{:}\kappa.\ ([R\uparrow/k+1]S) & \\
[R/k](S\ T) & = ([R/k]S)\ ([R/k]T) & \\
[R/k](S\ [T]) & = ([R/k]S)\ ([R/k]T) & \\
[R/k](S \to T) & = ([R/k]S) \to ([R/k]T) & \\
[R/k]* & = * &
\end{array}
$$

Variable lookup. $\Gamma(k)$ berechnet aus dem de Bruijn-Index k den zu k gehörenden Eintrag im Kontext Γ

$$
\begin{array}{ll}
(\Gamma, x{:}T)(0) & = T \uparrow \\
(\Gamma, x = R{:}T)(0) & = R \uparrow{:}\ T \uparrow \\
(\Gamma, x{:}T)(k+1) & = \Gamma(k) \uparrow \\
(\Gamma, x = R{:}T)(k+1) & = \Gamma(k) \uparrow
\end{array}
$$

$(R{:}T) \uparrow$ steht für $(R \uparrow{:}\ T \uparrow)$

Schwache Kopfnormalform. Für die Typ-Prüfung benötigt man eine Funktion, die schwache Kopfnormalformen von Termen berechnet. Ein Term V ist

dann in schwacher Kopfnormalform, wenn er keinen Kopf-Redex besitzt, d.h. $V \neq (\lambda X\!:\!\kappa.\ F)G_0\vec{G}$

$$\frac{\Gamma(k) = F\!:\!\kappa \qquad \Gamma \vdash F\ \vec{G} \searrow V}{\Gamma \vdash k\vec{G} \searrow V} \qquad \frac{[G_0/0]F\ \vec{G} \searrow V}{(\lambda X\!:\!\kappa.\ F)G_0\ \vec{G} \searrow V}$$

Alle anderen Terme sind bereits in schwacher Kopfnormalform.

3.2.2 Kind Checking

Das Kind-Checking besteht aus zwei (wechselseitig rekursiven) Funktionen: Der Kind-Prüfung und der Kind-Synthese.

Kind-Prüfung: $\Gamma \vdash F :\Leftarrow \kappa$ Die Kind-Prüfung konstruiert aus F den Kind κ' von F und vergleicht diesen mit dem angegebenen Kind κ. Bei Gleichheit von κ und κ' ist die Kind-Prüfung erfolgreich.

$$\frac{\Gamma \vdash F :\Rightarrow \kappa' \qquad \kappa' = \kappa}{\Gamma \vdash F :\Leftarrow \kappa}$$

Kind-Synthese $\Gamma \vdash F :\Rightarrow \kappa$ Die Kind-Synthese berechnet den Kind κ eines Typs F. Beispielsweise ist der Kind einer Abstraktion ein Funktionskind, und zwar von Kind κ_1, dem Kind der Abstraktionsvariablen, in den Kind κ_2, dem Kind des Rückgabetyps F. Der Kind einer Variable ist immer der Kind, mit dem sie annotiert wurde.

$$\frac{\Gamma(k) = \kappa \text{ or } \Gamma(k) = (A\!:\!\kappa)}{\Gamma \vdash k :\Rightarrow \kappa} \qquad \frac{\Gamma \vdash A :\Leftarrow * \qquad \Gamma \vdash B :\Leftarrow *}{\Gamma \vdash A \to B :\Rightarrow *} \qquad \frac{\Gamma, X\!:\!\kappa \vdash A :\Leftarrow *}{\Gamma \vdash \forall X\!:\!\kappa.\ A :\Rightarrow *}$$

$$\frac{\Gamma, X\!:\!\kappa_1 \vdash F :\Rightarrow \kappa 2}{\Gamma \vdash \lambda X\!:\!\kappa_1.\ F :\Rightarrow \kappa_1 \to \kappa_2} \qquad \frac{\Gamma \vdash F :\Rightarrow \kappa_1 \to \kappa_2 \qquad \Gamma \vdash G :\Leftarrow \kappa_1}{\Gamma \vdash F\ G :\Rightarrow \kappa_2}$$

3.2.3 Algorithmische Gleichheit

Schwierigkeiten bei der Typ-Prüfung im System F_ω
Im System F_ω gibt es, durch Applikation und Abstraktion auf Typ-Ebene, unzählige Möglichkeiten, ein und denselben Typ zu notieren. Beispiel: Sei

$$\mathit{Prod} = \lambda X \lambda Y.\ \forall Z.\ (X \to Y \to Z) \to Z$$

Dann stehen folgende Ausdrücke für denselben Typ:

$$Prod\ Int\ Int = \forall Z.\ (Int \to Int \to Z) \to Z$$

Man benötigt daher einen Algorithmus, der die Gleichheit zweier Typen entscheidet.

Algorithmische Gleichheit.
Um zu testen, ob zwei Typen F und F' gleich sind, könnten wir einfach beide Typen vollständig normalisieren und auf syntaktische Gleichheit prüfen. Wir verwenden die feinere Methode von Coquand [3], die jeweils nur die schwache Kopfnormalform bildet, dann die Köpfe vergleicht, und nur bei Gleichheit der Köpfe auch die Argumente schwach kopfnormalisiert und weiter auf Gleichheit prüft.

Unsere Implementierung dieses Prinzips beruht auf zwei wechselseitig rekursiven Funktionen: Der strukturellen Gleichheit $\Gamma \vdash F = F' :\Rightarrow \kappa$ und der Kind-gesteuerten Gleichheit $\Gamma \vdash F = F' :\Leftarrow \kappa$.

Kind-gesteuerte Gleichheit $\Gamma \vdash F = F' :\Leftarrow \kappa$
Falls die zu vegleichenden Typen einen Funktionskind besitzen, entfernen wir diesen durch (evtl. mehrfache) Applikation einer beliebigen, „frischen“ Variable (in de Bruijn-Notation einfach die Variable mit Index 0). Handelt es sich um echte Typen, werden diese schwach kopfnormalisiert und dann auf strukturelle Gleichheit überprüft.

$$\frac{\Gamma, X:\kappa \vdash (F\uparrow)0 = ((F'\uparrow)0 :\Leftarrow \kappa'}{\Gamma \vdash F = F' :\Leftarrow \kappa \to \kappa'} \qquad \frac{F \searrow V \qquad F' \searrow V' \qquad \Gamma \vdash V = V' :\Rightarrow *}{\Gamma \vdash F = F' :\Leftarrow *}$$

Strukturelle Gleichheit $\Gamma \vdash F = F' :\Rightarrow \kappa$
Bei der Überprüfung auf strukturelle Gleichheit zweier Typen muss man beachten, dass bestimmte Teile der Typen noch nicht kopfnormalisiert wurden. Daher muss man diese Teile auf Kind-gesteuerte Gleichheit überprüfen. Dies ist beispielsweise der Fall bei Applikationen, wo die Argument-Typen einer Kopfnormalisierung unterzogen werden müssen.

$$\frac{\Gamma(k) = \kappa \text{ or } \Gamma(k) = (F:\kappa)}{\Gamma \vdash k = k :\Rightarrow \kappa} \qquad \frac{\Gamma \vdash F = F' :\Rightarrow \kappa \to \kappa' \qquad \Gamma \vdash G = G' :\Leftarrow \kappa}{\Gamma \vdash F\ G = F'\ G' :\Rightarrow \kappa'}$$

$$\frac{\Gamma \vdash A = A' :\Leftarrow * \qquad \Gamma \vdash B = B' :\Leftarrow *}{\Gamma \vdash A \to B = A' \to B' :\Rightarrow *} \qquad \frac{\Gamma, X:\kappa \vdash A = A' :\Leftarrow *}{\Gamma \vdash \forall X:\kappa.\ A = \forall X':\kappa'.\ A' :\Rightarrow *}$$

3.2.4 Type Checking

Nachdem nun die Gleichheit zweier Typen definiert wurde, wollen wir die Aussage treffen, ob ein Term e vom Typ C ist. Hierzu benötigen wir zunächst eine Funtion, die den Typ eines Terms berechnet.

Typ-Synthese $\Gamma \vdash e :\Rightarrow A$
Die Typ-Synthese liefert den Typ eines Terms e. Sie folgt dabei den u.a. Regeln. Eine Variable beispielsweise ist immer von dem Typ, mit dem sie annotiert wurde. Der Typ einer Applikation $f\ e$ entspricht, unter der Voraussetzung, dass die Funktion f vom Pfeiltyp $A \to B$ ist und das Argument e vom Typ A, dem Ergebnistyp B der Funktion f.

$$\frac{\Gamma(k) = A \text{ or } \Gamma(k) = (e : A)}{\Gamma \vdash k :\Rightarrow A} \qquad \frac{\Gamma \vdash e :\Rightarrow A_0 \qquad \Gamma \vdash A_0 \searrow \forall X : \kappa.\ A \qquad \Gamma \vdash e :\Leftarrow \kappa}{\Gamma \vdash e\ [F] :\Rightarrow ([F \uparrow /0]A) \downarrow}$$

$$\frac{\Gamma, X : \kappa \vdash e :\Rightarrow A}{\Gamma \vdash \Lambda X : \kappa.\ e :\Rightarrow \forall X : \kappa.\ A} \qquad \frac{\Gamma \vdash A :\Leftarrow * \qquad \Gamma, x : A \vdash e :\Rightarrow B}{\Gamma \vdash \lambda x : A.\ e :\Rightarrow A \to (B \downarrow)}$$

$$\frac{\Gamma \vdash f :\Rightarrow A_0 \qquad \Gamma \vdash A_0 \searrow A \to B \qquad \Gamma \vdash e :\Leftarrow A}{\Gamma \vdash f\ e :\Rightarrow B}$$

Typ-Prüfung $\Gamma \vdash e :\Leftarrow C$
Die eigentliche Typ-Prüfung berechnet den Typ A eines Terms e und überprüft, ob dieser mit dem angegebenen Typ C übersinstimmt (unter Benutzung der von uns definierten algorithmischen Gleichheit)

$$\frac{\Gamma \vdash e :\Rightarrow A \qquad \Gamma \vdash A = C :\Leftarrow *}{\Gamma \vdash e :\Leftarrow C}$$

Kapitel 4

Funktionsweise des Typ-Checkers

In diesem Kapitel wird die Umgangsweise mit dem Typ-Checker erklärt, welcher im Rahmen dieser Arbeit erstellt wurde.

4.1 Funktionsweise

Das Programm fomega erhält als Argumente einen oder mehrere Dateinamen, welche dem unten angegebene Dateiformat genügen müssen (die Dateiendung .church ist allerdings nicht zwingend).

Befindet sich in einer Datei ein Fehler bezüglich der Syntax, wird der Parser dieses bemerken und mit einer Fehlermeldung darauf hinweisen. Wenn sich kein syntaktischer Fehler in der Datei befindet, wird für alle deklarierten Terme bzw. Typen eine Typ- bzw. Kind-Prüfung durchgeführt, d.h. es wird überprüft, ob der für den deklarierten Term angegebene Typ mit dem tatsächlichen Typ übereinstimmt. Ist dies nicht der Fall, wird in einer Fehlermeldung auf den Fehler hingewiesen.

Deklarationen sind nicht dateiübergreifend. Eine Deklaration in „file1.church“ ist also in „file2.church“ nicht sichtbar.

4.2 Syntax

Die Syntax für die Eingabe kann man der folgenden Tabelle entnehmen:

$t ::=$		Terme:
	x	Variable
	$\backslash x : T.\ t$	Abstraktion
	$t\ t$	Applikation
	$\backslash\backslash X : K.\ t$	Typ-Abstraktion
	$t\ [T]$	Typ-Applikation
	(t)	geklammerter Term
$T ::=$		Typen:
	X	Typvariable
	$T \rightarrow T$	Funktionstyp
	$forall\ X : K.\ T$	Universaltyp
	$\backslash X : K.\ T$	Operator-Abstraktion
	$T\ T$	Operator-Applikation
	(T)	geklammerter Typ
$K ::=$		Kinds:
	$*$	Kind für Proper Types
	$K \rightarrow K$	Kind für Operatoren
	(K)	geklammerter Kind

Es ist hierbei zu beachten, dass bei Typ- und Kindannotationen eine Klammerung zwingend notwendig ist, wenn es sich um eine Operator-Applikation oder einen Funktionstyp oder -kind handelt.

4.3 Dateiformat *.church*

Eine Datei, die mit diesem Typ-Checker geprüft werden soll, muss folgendem Format genügen:

Für jeden deklarierten Term „name“ muss folgender Eintrag enthalten sein:

$$term\ name : Typ = Term;$$

Für jeden deklarierten Typ „name“ muss folgender Eintrag enthalten sein:

$$type\ name : Kind = Typ;$$

Die Reihenfolge der Deklarationen ist von Bedeutung, da nur bereits deklarierte Terme und Typen in folgenden Deklarationen wiederverwendet werden dürfen! Das Layout der Datei ist dagegen völlig unerheblich, da „whitespaces“ ignoriert werden.

Beispieldatei

```
% Cartesian products

type prod         : * -> * -> *
                  = \A:* \B:* forall X:*. (A -> B -> X) -> X;

term pair         : forall A:* forall B:*. A -> B -> prod A B
                  = \\A:* \\B:* \a:A \b:B \\X:* \f:(A -> B -> X). f a b;

term fst          : forall A:* forall B:*. prod A B -> A
                  = \\A:* \\B:* \p:(prod A B). p[A] \a:A \b:B. a;

term snd          : forall A:* forall B:*. prod A B -> B
                  = \\A:* \\B:* \p:(prod A B). p[B] \a:A \b:B. b;
```

4.4 Quellcode

Dieser Typ-Checker wurde in Haskell implementiert. Wer sich den Quellcode des Typ-Checkers genauer anschauen möchte, findet ihn unter folgender Adresse zum Download:

Literaturverzeichnis

[1] Henk Barendregt. The type free lambda calculus. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter D.7, pages 1091–1132. North-Holland Publishing Company, 1977.

[2] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[3] Thierry Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

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