GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

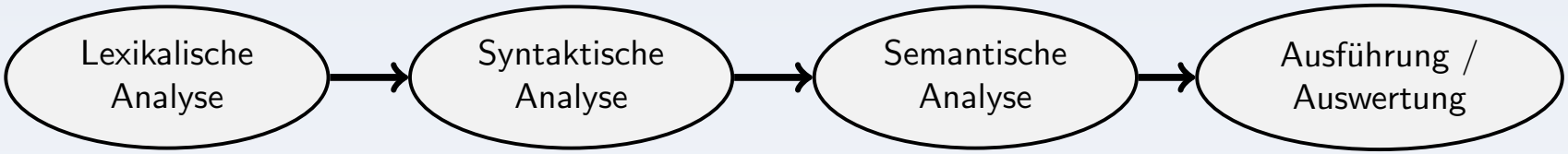
Praktikum BKSPP:

Blatt 4

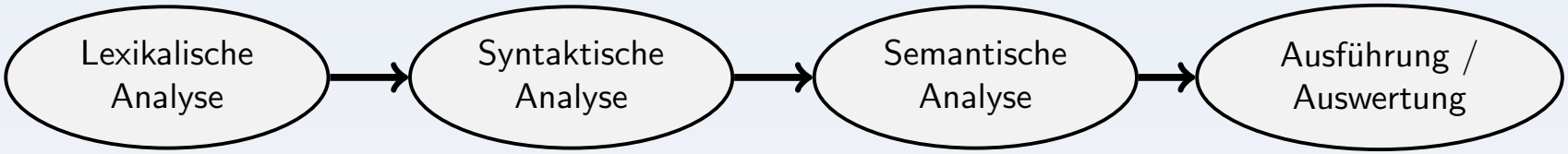
Dr. David Sabel

SoSe 2012

Interpreter allgemein

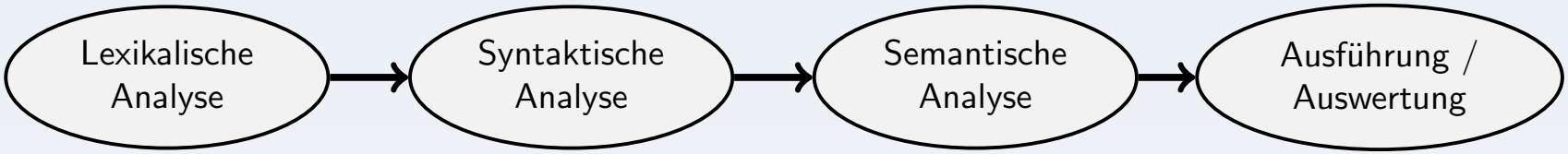


Interpreter allgemein



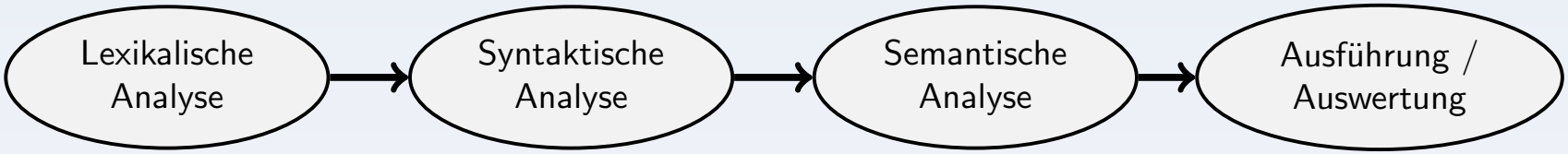
- **Lexikalische Analyse**: überführt Quelltext in Tokenstrom.
  (Entfernen von Zeilenkommentaren, Leerzeichen, Umbrüchen, etc.)
  Token: Syntaktische Einheit, z.B. 1000 nicht 1,0,0,0

Interpreter allgemein



- **Lexikalische Analyse**: überführt Quelltext in Tokenstrom.
  (Entfernen von Zeilenkommentaren, Leerzeichen, Umbrüchen, etc.)
  Token: Syntaktische Einheit, z.B. 1000 nicht 1,0,0,0
- **Syntaktische Analyse**: Prüft, ob Tokenstrom syntaktisch korrektes Programm (anhand e. kontextfreien Grammatik)
  Eingabe: Tokenstrom; Ausgabe: Syntaxbaum
  Programmierung von Hand aufwändig, deshalb Parsergeneratoren (für Haskell Happy)

Interpreter allgemein



- **Lexikalische Analyse**: überführt Quelltext in Tokenstrom.
  (Entfernen von Zeilenkommentaren, Leerzeichen, Umbrüchen, etc.)
  Token: Syntaktische Einheit, z.B. 1000 nicht 1,0,0,0
- **Syntaktische Analyse**: Prüft, ob Tokenstrom syntaktisch korrektes Programm (anhand e. kontextfreien Grammatik)
  Eingabe: Tokenstrom; Ausgabe: Syntaxbaum
  Programmierung von Hand aufwändig, deshalb Parsergeneratoren (für Haskell Happy)
- **Semantische Analyse**: Entdeckt Fehler, die nicht rein syntaktisch sind, z.B. ob alle Variablen deklariert sind, aber auch Typcheck

Aufgabenblatt

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Implementiere Interpreter für den Lambda-Kalkül

1. Lexikalische Analyse: Kommentare und Leerzeichen entfernen, Tokenstrom erzeugen
2. Syntaktische Analyse: Parser mit Happy generieren
3. Semantische Analyse: Typcheck bzgl. simple types
4. Ausführung: Call-by-name Auswertung implementieren

Der Lambda-Kalkül

Syntax

$$
\begin{array}{rcll}
e, e_i \in E := & x & & \text{Variable} \\
| & \lambda x.e & & \text{Abstraktion} \\
| & (e_1 \ @ \ e_2) & & \text{Anwendung / Applikation}
\end{array}
$$

- $\lambda x.e$: durch λx wird x im Rumpf e gebunden.
- Abstraktion sind anonyme Funktionen,
  z.B. $id(x) = x$ entspricht $\lambda x.x$.
  Argumente können Funktionen sein,
  z.B. $id(id)$ entspricht $(\lambda x.x) \ @ \ (\lambda x.x)$

Ziel des Lexers und Parsers

Überführe Lambda-Ausdrück in Haskell-interne Datenstruktur.

Lexikalische Analyse

Im Gegensatz zur formalen Syntax:

- \ statt λ im Quelltext
- Zeilenkommentare beginnend mit -- erlaubt
- Beliebige Leerzeichen, Zeilenumbrüche erlaubt
- Zusätzliche Klammerung erlaubt
- Variablennamen beginnen mit mit einem Buchstaben danach Buchstaben und Zahlen
- Variablen sind durch Leerzeichen getrennt: `"xy"` ist ein Token, aber `"x y"` sind zwei Token.

Lexikalische Analyse

Tokentyp

```
data Token = TokBackSl TokPos          -- '\'
           | TokDot    TokPos          -- '.'
           | TokBopen  TokPos          -- '('
           | TokBclose TokPos          -- ')'
           | TokApp    TokPos          -- '@'
           | TokVar    TokPos String -- Variablen
 deriving(Eq,Show)
```

Token speichern die Position im Quelltext:

```
type TokPos = (Int,Int) -- (Zeile,Spalte)
```

Aufgabe

Implementiere Funktion `lexInput :: String -> [Token]`
Hilfreich:

```
lexInput' String -> Int -> Int -> Token
lexInput' eingabe  zeile spalte = ...
```

Syntaktische Analyse

Datentyp für die interne Darstellung von Ausdrücken des Lambda-Kalküls:

```
data Expr  = Lambda VarName Expr
           | App Expr Expr
           | Var VarName
 deriving(Eq)

type VarName = String
```

Syntaktische Analyse

Datentyp für die interne Darstellung von Ausdrücken des Lambda-Kalküls:

```
data Expr  = Lambda VarName Expr
           | App Expr Expr
           | Var VarName
 deriving(Eq)

type VarName = String
```

Parser

- Happy-Überblick im Anhang
- Beispiel in `example.y`
- Parser (`Parser.hs`) erstellen: `happy Parser.ly`.
- Anschließend testen.

Parser (2)

Festlegungen, die der Parser beachten muss:

- Der Rumpf einer Abstraktion reicht so weit wie möglich
  $\lambda x.e_1 \ @ \ e_2$ ist vollgeklammert $\lambda x.(e_1 \ @ \ e_2)$ und nicht $((\lambda x.e_1) \ @ \ e_2)$.
- Die Anwendung @ ist links-assoziativ
  $e_1 \ @ \ e_2 \ @ \ e_3$ ist vollgeklammert $((e_1 \ @ \ e_2) \ @ \ e_3)$ und nicht $(e_1 \ @ \ (e_2 \ @ \ e_3))$
- Im Happy-Skript kann man Assoziativitäten und Prioritäten fest legen!

Aufbau eines Happy-Skripts (1)

Beispiel: Arithmetische Ausdrücke

- Tokentyp

```
data Token = TokenInt Int | TokenPlus | TokenMinus | TokenTimes
           | TokenDiv     | TokenOB   | TokenCB
```

- Typ des Syntaxbaums (Ausgabe des Parsers)

```
data Expr   = Plus Expr Expr | Minus Expr Expr | Times Expr Expr
            | Div Expr Expr  | NumberInt
```

Parserspezifikation: Dateiendung `.y` (oder `.ly`)

Aufbau eines Happy-Skripts (2)

1. Teil: Modulkopf (Haskell-Code) in geschweiften Klammern Z.B.

```
{
module Calc where
import Char
}
```

Aufbau eines Happy-Skripts (3)

2. Teil: Matchlist und Direktiven
Z.B.

```
%name calculator                -- Name des Parsers
%tokentype { Token }            -- Zuordnung des Tokentyps
%token                          -- Zurordnung Terminal in
                                -- der Grammatik zu Token
   int         { TokenInt $$ }
   '+'         { TokenPlus   }
   '-'         { TokenMinus  }
   '*'         { TokenTimes  }
   '/'         { TokenDiv    }
   '('         { TokenOB     }
   ')'         { TokenCB     }
%left '+' '-'                   -- left=linksassoziativ
%left '*' '/'                   -- Reihenfolge gibt Prioritaet an
                                -- niedrigste Prioritaet zuerst
```

Aufbau eines Happy-Skripts (4)

3. Teil: Grammatik

- Eingeleitet mit %%
- Typ des Syntaxbaums angeben
- Aktion in geschweiften Klammern hinter der Produktion

```
Expr :: { Expr }
Expr : Expr '+' Expr { Plus  $1 $3 }
     | Expr '-' Expr { Minus $1 $3 }
     | Expr '*' Expr { Times $1 $3 }
     | Expr '/' Expr { Div $1 $3   }
     | '(' Expr ')'  { $2          }
     | int           { Number $1   }
```

Aufbau eines Happy-Skripts (5)

4. Teil: Haskell-Code in geschweiften Klammern, sollte zumindest `happyError` definieren.

```
{
happyError :: [Token] -> a
happyError _ = error "parse error!"

data Token = TokenInt Int | TokenPlus | TokenMinus | TokenTimes
           | TokenDiv     | TokenOB   | TokenCB

data Expr  = Plus Expr Expr | Minus Expr Expr | Times Expr Expr
           | Div Expr Expr  | Number  Int

lexer :: String -> [Token]
lexer [] = []
lexer ('+':cs) = TokenPlus : lexer cs
lexer ('-':cs) = TokenMinus : lexer cs
lexer ('*':cs) = TokenTimes : lexer cs
lexer ('/':cs) = TokenDiv : lexer cs
lexer ('(':cs) = TokenOB : lexer cs
lexer (')':cs) = TokenCB : lexer cs
lexer (c:cs)
 | isSpace c = lexer cs
 | isDigit c = lexNum (c:cs)
 | otherwise = error ("error, can't lex symbol " ++ show "c")

lexNum cs = TokenInt (read num) : lexer rest
              where (num,rest) = span isDigit cs
}
```

Aufgabe

- Parser für den Lambda-Kalkül mit Happy erzeugen
- Parser sollte keine Shift-Reduce- oder Reduce-Reduce-Konflikte haben
- Pioritäten und Assoziativitäten verwenden!

Semantische Analyse

Überspringen wir zunächst erstmal zur Auswertung

Vorbereitung zur Auswertung

Bevor wir Auswerten können, brauchen wir

- α-Umbenennung, um umbenannte Kopien von Ausdrücken zu erstellen
- Substitution: $e_1[e_2/x] =$ ersetze alle x in e_1 durch frische Kopien von e_2

Vorbereitung zur Auswertung: α-Umbenennung

Freie und gebundene Variablen:

$$\begin{array}{lcllcl} FV(x) & = & x & GV(x) & = & \emptyset \\ FV(\lambda x.e) & = & FV(e) \setminus \{x\} & GV(\lambda x.e) & = & GV(e) \cup \{x\} \\ FV(e_1 \mathbin{@} e_2) & = & FV(e_1) \cup FV(e_2) & GV(e_1 \mathbin{@} e_2) & = & GV(e_1) \cup GV(e_2) \end{array}$$

Substitution: $e_1[e_2/x]$: Ersetze x durch e_2 in e_1

$$\begin{array}{lcl} x[e/x] & = & e \\ y[e/x] & = & y, \text{ falls } x \neq y \\ (\lambda y.e_1)[e_2/x] & = & \lambda y.(e_1[e_2/x]) \\ (e_1 \mathbin{@} e_2)[e_3/x] & = & (e_1[e_3/x] \mathbin{@} e_2[e_3/x]) \end{array}$$

Kontexte: Ausdruck mit Loch $[\cdot]$ an einer Stelle

$$C \in \mathbf{C} = [\cdot] \mid \lambda x.C \mid (C \mathbin{@} e) \mid (e \mathbin{@} C),$$

Einsetzung $C[e]$ ist der Ausdruck der entsteht, nachdem e an die Stelle des Lochs in C eingesetzt wurde

Vorbereitung zur Auswertung: α-Umbenennung

α-Umbenennungsschritt:

$$C[\lambda x.e] \xrightarrow{\alpha} C[\lambda y.e[y/x]] \text{ falls } y \notin GV(\lambda x.e) \cup FV(\lambda x.e)$$

- Die reflexiv-transitive Hülle solcher α-Umbenennungen heißt α-Äquivalenz.
- α-äquivalente Ausdrücke werden als gleich angesehen, z.B. $\lambda x.x =_\alpha \lambda y.y$

Ausdruck erfüllt DVC gdw. alle gebundenen Variablen haben unterschiedliche Namen und Namen gebundener Variablen stets verschieden sind von Namen freier Variablen.

Aufgabe: Umbenennung

Im Modul `Util`, die Funktion

```
renameExpr :: [VarName] -> Expr -> (Expr,[VarName])
```

- erhält: Liste von frischen Variablennamen und Ausdruck
- liefert: Umbenannten Ausdruck + restliche Variablen

Tipps: Man muss sich die Umbenennung am Binder merken

```
renameExpr' (n:namen) λx.e merkliste =
 renameExpr' namen e (x,n):merkliste

renameExpr' (n:namen) x+ merkliste =
 case lookup x merkliste of
   Just y -> y
   Nothing -> x
```

Aufgabe: Substitution

Im Modul `Util` eine Funktion

```
substitute :: [VarName] -> VarName -> Expr -> Expr -> (Expr,[VarName])
```

- erhält: frische Namen, eine Variable x und zwei Ausdrücke e_1, e_2
- liefert: $e_2[e_1/x]$ (wobei jedes e_1 vorher umbenannt wird) und verbleibende Namen

Zum Umbenennen `renameExpr` verwenden.

Beispiel:
`substitute ["a","b","c",...]` $z\ (\lambda x.\lambda y.x)\ (\lambda w.(z\ @\ z))$
ergibt

$$(\lambda w.(z[\lambda x.\lambda y.x/z]\ @\ z[\lambda x.\lambda y.x/z]))$$

Aufgabe: Substitution

Im Modul `Util` eine Funktion

```
substitute :: [VarName] -> VarName -> Expr -> Expr -> (Expr,[VarName])
```

- erhält: frische Namen, eine Variable x und zwei Ausdrücke e_1, e_2
- liefert: $e_2[e_1/x]$ (wobei jedes e_1 vorher umbenannt wird) und verbleibende Namen

Zum Umbenennen `renameExpr` verwenden.

Beispiel:
`substitute ["a","b","c",...]` $z\ (\lambda x.\lambda y.x)\ (\lambda w.(z\ @\ z))$
ergibt

$$(\lambda w.(z[\lambda a.\lambda b.a/z]\ @\ z[\lambda x.\lambda y.x/z]))$$

Aufgabe: Substitution

Im Modul `Util` eine Funktion

```
substitute :: [VarName] -> VarName -> Expr -> Expr -> (Expr,[VarName])
```

- erhält: frische Namen, eine Variable x und zwei Ausdrücke e_1, e_2
- liefert: $e_2[e_1/x]$ (wobei jedes e_1 vorher umbenannt wird) und verbleibende Namen

Zum Umbenennen `renameExpr` verwenden.

Beispiel:
`substitute ["a","b","c",...]` $z\ (\lambda x.\lambda y.x)\ (\lambda w.(z\ @\ z))$
ergibt

$$(\lambda w.(z[\lambda a.\lambda b.a/z]\ @\ z[\lambda c.\lambda d.c/z]))$$

Aufgabe: Substitution

Im Modul `Util` eine Funktion

```
substitute :: [VarName] -> VarName -> Expr -> Expr -> (Expr,[VarName])
```

- erhält: frische Namen, eine Variable x und zwei Ausdrücke e_1, e_2
- liefert: $e_2[e_1/x]$ (wobei jedes e_1 vorher umbenannt wird) und verbleibende Namen

Zum Umbenennen `renameExpr` verwenden.

Beispiel:
`substitute ["a","b","c",...]` $z\ (\lambda x.\lambda y.x)\ (\lambda w.(z\ @\ z))$
ergibt

$$(\lambda w.((\lambda a.\lambda b.a)\ @\ (\lambda c.\lambda d.c)))$$

Operationale Semantik

- Operationale Semantik legt fest, wie man ein Programm ausführt.
- In imperativen Sprachen: Wie verändert jeder Befehl den Zustand
- Im Lambda-Kalkül: Auswerten statt ausführen
- Wesentliche Regel: Auswertung der Funktionsanwendung auf Argumente:

$$(\beta) \qquad (\lambda x.e_1) \mathbin{@} e_2 \to e_1[e_2/x]$$

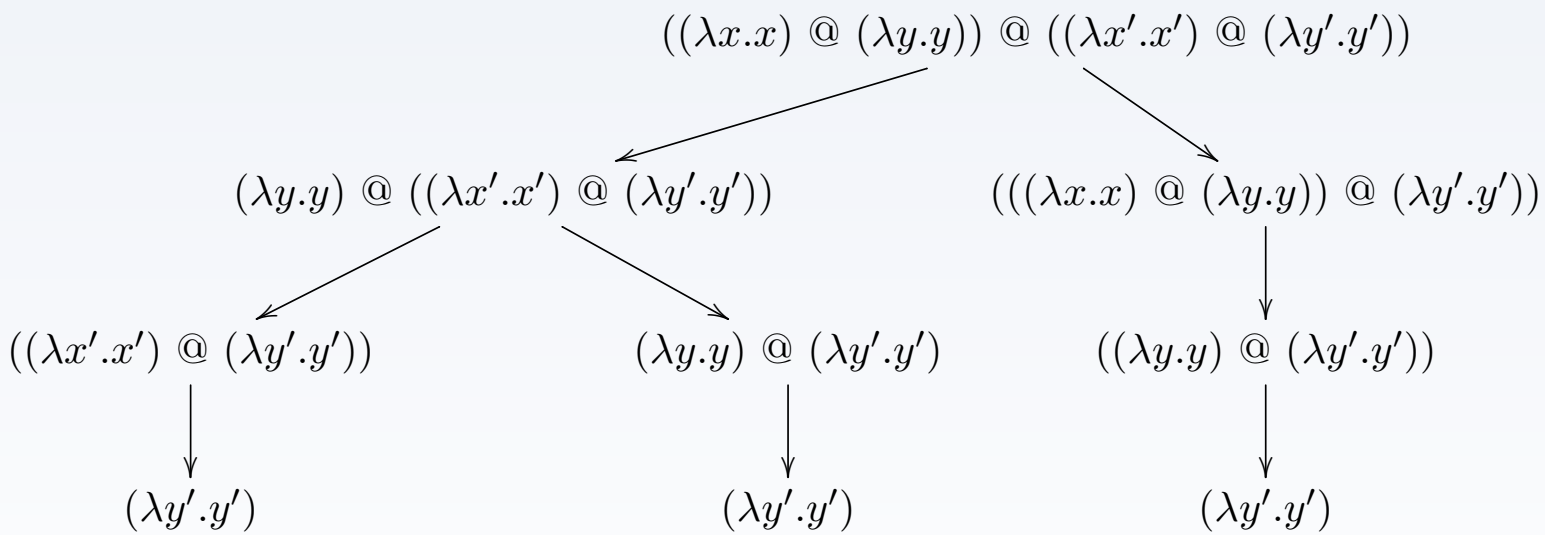
- Beispiel: $(\lambda x.\lambda y.x) \mathbin{@} (\lambda z.z) \xrightarrow{\beta} \lambda y.\lambda z.z$

Reduktionsstrategien

Anwendung der (β)-Reduktion auf Top-Level nicht immer möglich:

$$((\lambda x.x) \ @ \ (\lambda y.y)) \ @ \ ((\lambda x'.x') \ @ \ (\lambda y'.y'))$$

Wo reduzieren? Verschiedene Möglichkeiten:



Reduktionsstrategie legt (eindeutig) fest, wo reduziert wird.

Call-by-Name Reduktion

- sucht immer den am weitesten oben und am weitesten links stehenden Redex
- Formale Definition mit (call-by-name) Reduktionskontexten

$$R ::= [\cdot] \mid (R \ @ \ \mathbf{Expr})$$

- Wenn $e_1 \xrightarrow{\beta} e_2$, dann ist $R[e_1] \xrightarrow{name} R[e_2]$ eine call-by-name Reduktion

Call-by-Name Reduktion

Leicht implementierbares Verfahren:

- Zustand: (Ausdruck, Stack) wobei Stack enthält Ausdrücke.
- Regeln auf dem Zustand:

$$\begin{array}{ll} \text{(push)} & ((e_1 \ @ \ e_2), S) \to (e_1, e_2 : S) \\ \text{(take)} & (\lambda x.e_1), e_2 : S) \to (e_1[e_2/x], S) \end{array}$$

- Wende Regeln solange an wie es geht.

Beispiel

	Ausdruck	Stack
	$(((\lambda x.\lambda y.x) \ @ \ ((\lambda w.w) \ @ \ (\lambda z.z))) \ @ \ (\lambda u.u))$	$[]$
$\xrightarrow{(push)}$	$((\lambda x.\lambda y.x) \ @ \ ((\lambda w.w) \ @ \ (\lambda z.z)))$	$[\lambda u.u]$
$\xrightarrow{(push)}$	$\lambda x.\lambda y.x$	$[((\lambda w.w) \ @ \ (\lambda z.z)), \lambda u.u]$
$\xrightarrow{(take)}$	$\lambda y.((\lambda w.w) \ @ \ (\lambda z.z))$	$[\lambda u.u]$
$\xrightarrow{(take)}$	$((\lambda w.w) \ @ \ (\lambda z.z))$	$[]$
$\xrightarrow{(push)}$	$\lambda w.w$	$[\lambda z.z]$
$\xrightarrow{(take)}$	$\lambda z.z$	$[]$

Ergebnisse

Drei Möglichkeiten

- $(\lambda x.e, [])$, dann erfolgreich zu einer Abstraktion reduziert
- (x, S), dann freie Variable gefunden, nicht erfolgreich, aber stoppe
- $\rightarrow^{\omega}$, hält nicht

Call-by-Name Reduktion: Aufgabe

Im Modul `Evaluator` eine Funktion

```
evaluate :: [VarName] -> Expr -> Expr
```

- erhält frische Namen und Ausdruck
- liefert Ergebnis (Abstraktion)
- Fehlermeldung bei freier Variable
- oder hält nicht an
- Implementierung mit Stack (Stack kann durch Liste `[Expr]` darsgestellt werden)

Semantische Analyse: Einfaches Typsystem

- Syntax für einfache Typen:

$$\tau, \tau_i \in Typ ::= 0 \mid \tau_1 \to \tau_2$$

- Γ: Typumgebung, d.h. Abbildung von Variablen auf Typen
- $\Gamma \vdash e : \tau$ bedeutet: Unter der Typumgebung Γ ist der Typ τ für den Ausdruck e herleitbar.
- Typherleitungsregeln:

Anwendungsregel:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \text{ und } \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1 \ @ \ e_2) : \tau_2}$$

Abstraktionsregel:

$$\frac{\Gamma \cup \{x \mapsto \tau_1\} \vdash e : \tau_2}{\Gamma \vdash (\lambda x.e) : \tau_1 \to \tau_2}$$

Axiom für Variablen:

$$\frac{}{\Gamma \cup \{x \mapsto \tau\} \vdash x : \tau}$$

Beispiel

$$\dfrac{\dfrac{\dfrac{}{\{x \mapsto (0 \to 0)\} \vdash x : (0 \to 0)}}{\emptyset \vdash \lambda x.x : (0 \to 0) \to (0 \to 0)} \text{ und } \dfrac{\dfrac{}{\{y \mapsto 0\} \vdash y : 0}}{\emptyset \vdash \lambda y.y : 0 \to 0}}{\emptyset \vdash (\lambda x.x) \ @ \ (\lambda y.y) : 0 \to 0}$$

Beispiel

$$\dfrac{\dfrac{\dfrac{}{\{x \mapsto (0 \to 0)\} \vdash x : (0 \to 0)}}{\emptyset \vdash \lambda x.x : (0 \to 0) \to (0 \to 0)} \text{ und } \dfrac{\dfrac{}{\{y \mapsto 0\} \vdash y : 0}}{\emptyset \vdash \lambda y.y : 0 \to 0}}{\emptyset \vdash (\lambda x.x) \ @ \ (\lambda y.y) : 0 \to 0}$$

- Aber hier nur richtig geraten!
- Regel für Abstraktion ist nicht richtig algorithmisch:
- Man muss den Typ schon kennen, damit man richtig rechnet.
- Deswegen: Algorithmischeres Verfahren

Erweitertes Verfahren

- Erweiterung um Variablen α:

$$\tau, \tau_i ::= 0 \mid \tau_1 \to \tau_2 \mid \alpha$$

- Und um Typgleichungen E:
  Menge von Gleichungen der Form $\tau_1 =^? \tau_2$
- $\Gamma \vdash e : \tau, E$ bedeutet
  Unter der Typannahme Γ ist e typisierbar mit $\sigma(\tau)$, wenn σ die Lösung der Gleichungen in E ist

Typregeln mit Gleichungen

Anwendungsregel:

$$\frac{\Gamma \vdash e_1 : \tau_1, E_1 \text{ und } \Gamma \vdash e_2 : \tau_2, E_2}{\Gamma \vdash (e_1 \ @ \ e_2) : \alpha, \{\tau_1 =^? \tau_2 \to \alpha\} \cup E_1 \cup E_2} \ \alpha \text{ neu}$$

Abstraktionsregel:

$$\frac{\Gamma \cup \{x : \alpha\} \vdash e : \tau, E}{\Gamma \vdash (\lambda x.e) : \alpha \to \tau, E}$$

Axiom für Variablen:

$$\frac{}{\Gamma \cup \{x \mapsto \tau\} \vdash x : \tau, \emptyset}$$

Leiten jedoch noch keinen Typ her, sondern Typ + Gleichungssystem

Typregeln mit Gleichungen

Anwendungsregel:

$$\frac{\Gamma \vdash e_1 : \tau_1, E_1 \text{ und } \Gamma \vdash e_2 : \tau_2, E_2}{\Gamma \vdash (e_1 \ @ \ e_2) : \alpha, \{\tau_1 =^? \tau_2 \to \alpha\} \cup E_1 \cup E_2} \ \alpha \text{ neu}$$

Abstraktionsregel:

$$\frac{\Gamma \cup \{x : \alpha\} \vdash e : \tau, E}{\Gamma \vdash (\lambda x.e) : \alpha \to \tau, E}$$

Axiom für Variablen:

$$\frac{}{\Gamma \cup \{x \mapsto \tau\} \vdash x : \tau, \emptyset}$$

Leiten jedoch noch keinen Typ her, sondern Typ + Gleichungssystem

Typherleitung

$$\frac{\emptyset \vdash e : \tau, E \text{ und } \gamma \text{ ist Lösung von } E}{\emptyset \vdash e : \rho(\gamma(\tau))}$$

für jede Grundsubstitution ρ

Typregeln mit Gleichungen

Anwendungsregel:

$$\frac{\Gamma \vdash e_1 : \tau_1, E_1 \text{ und } \Gamma \vdash e_2 : \tau_2, E_2}{\Gamma \vdash (e_1 \ @ \ e_2) : \alpha, \{\tau_1 =^? \tau_2 \to \alpha\} \cup E_1 \cup E_2} \ \alpha \text{ neu}$$

Abstraktionsregel:

$$\frac{\Gamma \cup \{x : \alpha\} \vdash e : \tau, E}{\Gamma \vdash (\lambda x.e) : \alpha \to \tau, E}$$

Axiom für Variablen:

$$\frac{}{\Gamma \cup \{x \mapsto \tau\} \vdash x : \tau, \emptyset}$$

Leiten jedoch noch keinen Typ her, sondern Typ + Gleichungssystem

Typherleitung

$$\frac{\emptyset \vdash e : \tau, E \text{ und } \gamma \text{ ist Lösung von } E}{\emptyset \vdash e : \rho(\gamma(\tau))}$$

für jede Grundsubstitution ρ
Grundsubstitution uninteressant, d.h. wir berechnen nur $\gamma(\tau)$

Unifikationsalgorithmus

- Eingabe: Gleichungssystem E
- Gesucht: Lösung für E, d.h. Abbildung von Typvariablen auf Typen, so dass alle Gleichung links und rechts gleich sind
- Datenstruktur: (G, E), G: Gelöste Gleichungen, E Ungelöste Gleichungen

Löschregel $(G, E \cup \{\tau =^? \tau\}) \to (G, E)$, wobei τ eine Typvariable oder $\tau = 0$

Dekomposition $(G, E \cup \{\tau_1 \to \tau_2 =^? \tau_3 \to \tau_4\}) \to (G, E \cup \{\tau_1 =^? \tau_3, \tau_2 =^? \tau_4)$

Gelöst $(G, E \cup \{\alpha =^\tau\}) \to (G[\tau/\alpha] \cup \{\alpha =^\tau\}, E[\tau/\alpha])$

wobei α Typvariable und α kommt nicht in τ vor

Occurs Check $(G, E \cup \{\alpha =^? \tau\}) \to \mathit{Fail}$, falls α in τ vorkommt

Fail $(G, E \cup \{\tau_1 \to \tau_2 =^? 0\}) \to \mathit{Fail}$

Fail $(G, E \cup \{0 =^? \tau_1 \to \tau_2\}) \to \mathit{Fail}$

Orientierung $(G, E \cup \{\tau =^? \alpha\}) \to (G, E \cup \{\alpha =^? \tau\})$

wobei α Typvariable, τ keine Typvariable

Beispiel

$$\dfrac{\dfrac{\overline{\{x:\alpha_1\}\vdash x:\alpha_1,\emptyset}}{\emptyset\vdash\lambda x.x:\alpha_1\to\alpha_1,\emptyset} \text{ und } \dfrac{\overline{\{y:\alpha_2\}\vdash y:\alpha_2,\emptyset}}{\emptyset\vdash\lambda y.y:\alpha_2\to\alpha_2,\emptyset}}{\emptyset\vdash(\lambda x.x)\ @\ (\lambda y.y):\alpha_3,\{\alpha_1\to\alpha_1 =^? (\alpha_2\to\alpha_2)\to\alpha_3\}}$$

Löse $\{\alpha_1\to\alpha_1 =^? (\alpha_2\to\alpha_2)\to\alpha_3\}$:

G	E	nächste verwendete Regel
	$\alpha_1\to\alpha_1 =^? (\alpha_2\to\alpha_2)\to\alpha_3$	(Dekomposition)
	$\alpha_1 =^? \alpha_2\to\alpha_2$ $\alpha_1 =^? \alpha_3$	(Gelöst)
$\alpha_1 =^? \alpha_3$	$\alpha_3 =^? \alpha_2\to\alpha_2$	(Gelöst)
$\alpha_1 =^? \alpha_2\to\alpha_2$ $\alpha_3 =^? \alpha_2\to\alpha_2$		

Ergibt: $\gamma = \{\alpha_1\mapsto\alpha_2\to\alpha_2, \alpha_3\mapsto\alpha_2\to\alpha_2\}$ und:

$$\gamma(\alpha_3) = \alpha_2\to\alpha_2$$

Weiteres Beispiel

$$\dfrac{\dfrac{}{\{x \mapsto \alpha_1\} \vdash x : \alpha_1, \emptyset} \text{ und } \dfrac{}{\{x \mapsto \alpha_1\} \vdash x : \alpha_1, \emptyset}}{\dfrac{\{x \mapsto \alpha_1\} \vdash (x \ @ \ x) : \alpha_2, \{\alpha_1 =^? \alpha_2 \to \alpha_1\}}{\emptyset \vdash \lambda x.(x \ @ \ x) : \alpha_1 \to \alpha_2, \{\alpha_1 =^? \alpha_2 \to \alpha_1\}}}$$

Unifikation

G	E	nächste verwendete Regel
	$\alpha_1 =^? \alpha_2 \to \alpha_1$	(Occurs Check)
Fail		

D.h. $\lambda x.(x \ x)$ ist nicht typisierbar.

Implementierung

Darstellung der Typen in Haskell:

```
data Type = Unit                  -- 0
          | Type :-> Type         -- t1 -> t2
          | TVar TypVarName       -- Typvariable
 deriving (Eq)

type TypVarName = String

type Umgebung  = [(TypVarName, Type)]
type Gleichung = (Type,Type)
```

Implementierung Unifikation

Im Modul `TypeCheck` eine Funktion

```
unify :: [Gleichung] -> [Gleichung] -> Maybe [Gleichung]
```

wobei `unify` G und E erwartet und entweder fehlschlägt (`Nothing`) oder das gelöste Gleichungssystem liefert.

Hilfsfunktion für die Ersetzung $F[\tau/\alpha]$ vom Typ

```
ersetze :: TypVarName -> Type -> [Gleichung] -> [Gleichung]
```

Implementierung Typcheck

Im Modul `TypeCheck`:

```
tcheck :: [TypVarName]
          -> Umgebung
          -> Expr
          -> (Type,[Gleichung],[TypVarName])
```

- erhält: frische Namen für Typvariablen, Typumgebung, Ausdruck
- liefert: Typ, Gleichungen, restliche Namen

Typregeln von unten nach oben rekursiv Programmieren.
Hauptfunktion `runTCheck :: Expr -> Type` ist schon vorgegeben

- Ruft `tcheck` und `unify` auf
- Wendet Substitution an

Letzte Aufgabe: Interpreter

Konsolenprogramm, das

- Lext und parst
- Typcheck durchführt
- Ausdruck auswertet

Option einfügen, so dass Typcheck ausgelassen wird.