GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Grundlagen der Programmierung 3 A

Typen, Typberechnung und Typcheck

Prof. Dr. Manfred Schmidt-Schauß

Sommersemester 2017

Haskell, Typen, und Typberechnung

Ziele:

- Haskells Typisierung
- Typisierungs-Regeln
- Typ-Berechnung
- Milners Typcheck (Robin Milner)

Einige andere Programmiersprachen

- ab **Java 5**: generische Typen verwandt mit polymorphen Typen
- *ML* (und *CAML*, *OCAML*) hat parametrisch polymorphes Typsystem
- **JavaScript** Nachteile: Es fehlt eine statische Typisierung
- **TypeScript** (Microsoft): Neue JavaScript Variante mit statischer Typisierung und Typinferenz.
- Google Ankündigung: **Angular 2** (JavaScript-Webframework) zukünftig auf Basis von TypeScript

Typisierung in Haskell

Haskell hat eine *starke, statische Typisierung*.
mit parametrisch polymorphen Typen.

- jeder Ausdruck muss einen Typ haben
- Der Typchecker berechnet Typen aller Ausdrücke und prüft Typen zur Compilezeit
- Es gibt keine Typfehler zur Laufzeit
  d.h. kein dynamischer Typcheck nötig.

Überladung und Konversion in Haskell

Haskell: keine Typkonversion

Es gibt Überladung:
z.B. arithmetische Operatoren: $+, -.*, /$

Zahlkonstanten für ganze Zahlen sind überladen

Typcheck erstmal ohne Überladung
ohne Kinds
und ohne Typen höherer Ordnung

Typisierung, Begriffe

polymorph: Ein Ausdruck kann viele Typen haben (vielgestaltig, polymorph).

parametrisch polymorph: Die (i.a. unendliche) Menge der Typen entspricht einem schematischen Typausdruck

Beispiel `length`

Schema: $[a]$`->Int`

Instanzen: `[Int]->Int`, `[Char]->Int`, `[[Char]]->Int` usw.

Syntax von Typen in Haskell (ohne Typklassen)

$\langle \text{Typ} \rangle$ ::= $\langle \text{Basistyp} \rangle \mid (\langle \text{Typ} \rangle) \mid \langle \text{Typvariable} \rangle$
$\mid \langle \text{Typkonstruktor}_n \rangle \{\langle \text{Typ} \rangle\}^n$
(n = Stelligkeit)

$\langle \text{Basistyp} \rangle$ ::= `Int` | `Integer` | `Float` | `Rational` | `Char` | ...

Syntax von Typen

Typkonstruktoren können benutzerdefiniert sein (z.B. `Baum` a)

Vordefinierte Typkonstruktoren:	Liste	$[\cdot]$	
	Tupel	`(.,...,.)`	
	Funktionen	$\cdot \to \cdot$	(Stelligkeit 2, Infix)

Konvention zu *Funktionstypen* mit $\to$
$a \to b \to c \to d$ bedeutet: $a \to (b \to (c \to d))$.

Interpretation der Typen

```
length :: [a] -> Int
```

Interpretation: Für alle Typen `typ` ist `length` eine Funktion, die vom Typ `[typ]` $\rightarrow$ `Int` ist.

Komposition

Beispiel: Komposition von Funktionen:

```
komp::(a -> b) -> (c -> a) -> c -> b
komp f g x = f (g x)
```

In Haskell ist `komp` vordefiniert und wird als „.“ geschrieben:
Beispielaufruf:

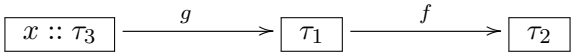
```
*Main> suche_nullstelle (sin . quadrat) 1 4 0.00000001
⏎
1.772453852929175
```

`(sin . quadrat)` entspricht $\texttt{sin}(x^2)$
und `quadrat . sin` entspricht $(\texttt{sin}(x))^2$.

Typ der Komposition

Erklärung zum Typ von `komp`, wobei {`a,b,c`} Typvariablen sind.
Ausdruck: f ‘`komp`‘ g bzw. f . g

`(a->b) ->`	`(c->a) ->`	`c->b`	
(τ_1 -> τ_2)			Typ von `(.)`
	(τ_3 -> τ_1)		Typ von f
		τ_3	Typ von g
			Typ des Arguments x der Komposition f . g
		τ_2	Typ des Resultats der Komposition $f(g\ x)$
		(τ_3 -> τ_2)	Typ von $(f\ .\ g)$



Typen von Konstruktoren

Typen von Konstruktoren werden durch deren `data`-Definition automatisch festgelegt!

```
data Baum a b = Empty | Blatt b
              | Knoten a (Baum a b) Baum a b)
```

Typen der Konstruktoren:

`Empty`	::	`Baum` $a\ b$
`Blatt`	::	$b \rightarrow$ `Baum` $a\ b$
`Knoten`	::	$b \rightarrow$ `Baum` $a\ b \rightarrow$ `Baum` $a\ b \rightarrow$ `Baum` $a\ b$

Typregeln

Wie berechnet man Typen von Ausdrücken?

Anwendung von Funktionsausdruck auf Argument

$$\frac{s :: \sigma \to \tau \ , \quad t :: \sigma}{(s\ t) :: ?}$$

Beispiele:

$$\frac{\texttt{quadrat} :: \texttt{Int} \to \texttt{Int} \quad , \quad 2 :: \texttt{Int}}{(\texttt{quadrat}\ 2) :: ?}$$

$$\frac{\dfrac{+ :: \texttt{Int} \to (\texttt{Int} \to \texttt{Int}) \quad , \quad 1 :: \texttt{Int}}{(1\ +) :: ?} \quad , \quad 2 :: \texttt{Int}}{((1\ +)\ 2) :: ?}$$

Typregel „Anwendung“: mehrere Argumente

$$\frac{s :: \sigma_1 \to \sigma_2 \ldots \to \sigma_n \to \tau \ , \quad t_1 :: \sigma_1 \ , \quad \ldots \ , \quad t_n :: \sigma_n}{(s\ t_1\ \ldots\ t_n) :: \tau}$$

Beispiele

$$\frac{+ :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}, \quad 1 :: \texttt{Int} \ , 2 :: \texttt{Int}}{(+\ 1\ 2) :: \texttt{Int}}$$

$$\frac{+ :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}}{(1 + 2) :: \texttt{Int}}$$

Beispiel

`even::Int -> Bool`	`a -> b` $\doteq$ `Int -> Bool`
`(.)::(a -> b) -> (c -> a) -> c -> b`	`c` $\doteq$ `Int`; `b` $\doteq$ `Bool`
`quadrat::Int -> Int`	`c -> a` $\doteq$ `Int -> Int`
`even . quadrat` :: `Int -> Bool`	

Anwendungsregel für polymorphe Typen

Ziel: Anwendung der Typregel für z.B. `length` oder `map`

Neuer Begriff: γ ist eine *Typsubstitution*
wenn sie Typen für Typvariablen einsetzt
$\gamma(\tau)$ nennt man Instanz von τ

Beispiel

Typsubstitution:	$\gamma = \{a \mapsto \texttt{Char}, b \mapsto \texttt{Float}\}$
Instanz:	$\gamma([a] \to \texttt{Int}) = [\texttt{Char}] \to \texttt{Int}$

Anwendungsregel für polymorphe Typen

$$\frac{s :: \sigma \to \tau, \quad t :: \rho \text{ und } \gamma(\sigma) = \gamma(\rho)}{(s\ t) :: \gamma(\tau)}$$

Berechnet den Typ von $(s\ t)$
wenn die Typen von s, t schon bekannt sind

Hierbei ist zu beachten:
Damit alle Freiheiten erhalten bleiben:

die Typvariablen in ρ müssen vorher umbenannt werden,
so dass σ und ρ keine gemeinsamen Typvariablen haben.

Beispiel zur polymorphen Anwendungsregel

Typ von `(map quadrat)` ?

$$\begin{array}{lll} \texttt{map} :: & (a \to b) & \to \quad ([a] \to [b]) \\ \texttt{quadrat} :: & \texttt{Int} & \to \quad \texttt{Int} \end{array}$$

Instanziiere mit: $\gamma = \{a \mapsto \texttt{Int}, b \mapsto \texttt{Int}\}$
ergibt: $\texttt{map} :: (\texttt{Int} \to \texttt{Int}) \to ([\texttt{Int}] \to [\texttt{Int}])$

Regelanwendung ergibt:

$$\frac{\texttt{map} :: (\texttt{Int} \to \texttt{Int}) \to ([\texttt{Int}] \to [\texttt{Int}]), \quad \texttt{quadrat} :: (\texttt{Int} \to \texttt{Int})}{(\texttt{map quadrat}) :: [\texttt{Int}] \to [\texttt{Int}]}$$

Polymorphe Regel für n Argumente

$$\frac{s :: \sigma_1 \to \sigma_2 \ldots \to \sigma_n \to \tau, \quad t_1 :: \rho_1, \ldots, t_n :: \rho_n \text{ und } \forall i : \gamma(\sigma_i) = \gamma(\rho_i)}{(s\ t_1 \ldots t_n) :: \gamma(\tau)}$$

Die Typvariablen in $\rho_1, \ldots, \rho_n$ müssen vorher umbenannt werden.

Beachte: verwende möglichst allgemeines γ
(kann berechnet werden; s.u.)

Wie berechnet man die Typsubstitutionen?

Unifikation: Berechnung der allgemeinsten Typsubstitution
im Typberechnungsprogramm bzw Typchecker

Unifikation wird benutzt im Typchecker von Haskell!

polymorphe Typen; Unifikation

$$\frac{s :: \sigma \to \tau, \quad t :: \rho \text{ und } \gamma \text{ ist } \textbf{allgemeinster Unifikator} \text{ von } \sigma \doteq \rho}{(s\ t) :: \gamma(\tau)}$$

(die Typvariablen in ρ müssen vorher umbenannt werden.)

Unifikation: Regelbasierter Algorithmus

Man braucht 4 Regeln, die auf $(E; G)$ operieren:

E: Menge von Typgleichungen,
G: Lösung; mit Komponenten der Form $x \mapsto t$.

Beachte: x, y ist Typvariable
s_i, t_i, s, t sind Typen,
f, g sind Typkonstruktoren

Unifikation: Die Regeln

Start mit $G = \emptyset$;
E: zu lösende Gleichung(en)

- $\dfrac{G;\ \{x \doteq x\} \cup E}{G; E}$
- $\dfrac{G;\ \{t \doteq x\} \cup E}{G; \{x \doteq t\} \cup E}$ Wenn t keine Variable ist
- $\dfrac{G;\ \{x \doteq t\} \cup E}{G[t/x] \cup \{x \mapsto t\}; E[t/x]}$ Wenn x nicht in t vorkommt
- $\dfrac{G;\ \{(f\ s_1\ \ldots\ s_n) \doteq (f\ t_1\ \ldots\ t_n)\} \cup E}{G; \{s_1 \doteq t_1, \ldots, s_n \doteq t_n)\} \cup E}$

Ersetzung: $E[t/x]$: alle Vorkommen von x werden durch t ersetzt
Effekt: $G[t/x]$: jedes $y \mapsto s$ wird durch $y \mapsto s[t/x]$ ersetzt

Unifikation: Regelbasierter Algorithmus

Fehlerabbruch, wenn:

- $x \doteq t$ in E, $x \neq t$ und x kommt in t vor. (Occurs-Check)
- $(f(\ldots) \doteq g(\ldots)$ kommt in E vor und $f \neq g$. (Clash)

Fehlerabbruch bedeutet: nicht typisierbar

Beispiel mit Typvariablen

Berechne Typ von `(map id)`

`map::`	$(a \to b)$	$\to ([a] \to [b])$
`id::`	$a' \to a'$	
Gesuchter Typ:		$\gamma([a] \to [b])$

Regelanwendung benötigt Lösung γ von $(a \to b) \doteq (a' \to a')$:

G	E
$\emptyset$;	$\{(a \to b) \doteq (a' \to a')\}$
$\emptyset$;	$\{a \doteq a', b \doteq a'\}$
$\{a \mapsto a'\}$;	$\{b \doteq a'\}$
$\{a \mapsto a', b \mapsto a'\}$;	$\emptyset$
$\{a \mapsto a', b \mapsto a'\}$;	

Einsetzen der Lösung $\gamma = \{a \mapsto a', b \mapsto a'\}$
in $[a] \to [b]$ ergibt:
`(map id)` :: $([a'] \to [a'])$.

Noch ein Beispiel

Berechne Typ von `(map head)`

`map::`	$(a \to b)$	$\to ([a] \to [b])$
`head::`	$[a] \to a$	
Gesuchter Typ:		$\gamma([a] \to [b])$

Regelanwendung benötigt Lösung γ von $(a \to b) \doteq ([a'] \to a')$:

G	E
$\emptyset$;	$\{(a \to b) \doteq ([a'] \to a')\}$
$\emptyset$;	$\{a \doteq [a'], b \doteq a'\}$
$\{a \mapsto [a']\}$;	$\{b \doteq a'\}$
$\{a \mapsto [a'], b \mapsto a'\}$;	$\emptyset$
$\{a \mapsto [a'], b \mapsto a'\}$;	

Einsetzen der Lösung $\gamma = \{a \mapsto [a'], b \mapsto a'\}$
in $[a] \to [b]$ ergibt:
`(map head)` :: $([[a']] \to [a'])$.

Beispiel `(foldr (:) [])` :: ??

```
foldr :: (a -> b -> b) -> b -> [a] -> b
(:) :: a -> [a] -> [a]    umbenannt: c -> [c] -> [c]
([]) :: [a]               umbenannt: [d]
```

G	E
$\emptyset$	$\{a \to b \to b \doteq c \to [c] \to [c], b \doteq [d]\}$
$\{b \mapsto [d]\}$	$\{a \to [d] \to [d] \doteq c \to [c] \to [c]\}$
$\{b \mapsto [d]\}$	$\{a \doteq c, [d] \doteq [c], [d] \doteq [c]\}$
$\{b \mapsto [d], a \mapsto c\}$	$\{[d] \doteq [c], [d] \doteq [c]\}$
$\{b \mapsto [d], a \mapsto c\}$	$\{d \doteq c, [d] \doteq [c]\}$
$\{b \mapsto [c], a \mapsto c, d \mapsto c\}$	$\{[c] \doteq [c]\}$
$\{b \mapsto [c], a \mapsto c, d \mapsto c\}$	$\{c \doteq c\}$
$\{b \mapsto [c], a \mapsto c, d \mapsto c\}$	$\{\}$

`(foldr (:) [])`$::[c] \to [c] \qquad = \gamma([a] \to b)$

Beispiel. Linksfaltung: `(foldl (:) [])`?

```
foldl :: (a -> b -> a) -> a -> [b] -> a
(:) :: a -> [a] -> [a]    umbenannt: c -> [c] -> [c]
([]) :: [a]               umbenannt: [d]
```

G	E
$\emptyset$	$\{a \to b \to a \doteq c \to [c] \to [c], a \doteq [d]\}$
$\{a \mapsto [d]\}$	$\{[d] \to b \to [d] \doteq c \to [c] \to [c]\}$
$\{a \mapsto [d]\}$	$\{[d] \doteq c, b \doteq [c], [d] \doteq [c]\}$
$\{a \mapsto [d], b \mapsto [c]\}$	$\{[d] \doteq c, [d] \doteq [c]\}$
$\{a \mapsto [d], b \mapsto [c]\}$	$\{c \doteq [d], [d] \doteq [c]\}$
$\{a \mapsto [d], b \mapsto [[d]], c \mapsto [d]\}$	$\{[d] \doteq [[d]]\}$
$\{a \mapsto [d], b \mapsto [[d]], c \mapsto [d]\}$	$\{d \doteq [d]\}$

nicht lösbar, da d in $[d]$ echt vorkommt

`(foldl (:) [])` ist nicht typisierbar!

Beispiel zu Typberechnung

Typ von `map length`

$$\frac{\texttt{map} :: (a \to b) \to ([a] \to [b]), \quad \texttt{length} :: [a'] \to \texttt{Int}}{(\texttt{map length}) :: ? = \gamma([a] \to [b])}$$

Unifiziere $(a \to b) \doteq [a'] \to \texttt{Int}$

G	E
$\emptyset$;	$\{(a \to b) \doteq ([a'] \to \texttt{Int})\}$
$\emptyset$;	$\{a \doteq [a'], b \doteq \texttt{Int}\}$
$\{a \mapsto [a']\}$;	$\{b \doteq \texttt{Int}\}$
$\{a \mapsto [a'], b \mapsto \texttt{Int}\}$;	$\emptyset$

Somit: $(\texttt{map length}) :: \gamma([a] \to [b]) = [[a']] \to [\texttt{Int}]$

Beispiele zu polymorpher Typberechnung

Berechne Typ der Liste $[1]$:

$$\frac{1 : \texttt{Int} \qquad (:) :: a \to [a] \to [a] \qquad [] :: [b]}{1 : [] ::?}$$

Anwendungsregel ergibt Gleichungen: $\{a \doteq \texttt{Int}, [a] \doteq [b]\}$

Lösung: $\gamma = \{a \mapsto \texttt{Int}\}$

Typ $(1 : []) :: [\texttt{Int}]$

Beispiel zu Typfehler

$[1, \ \texttt{'a'}]$ hat keinen Typ:

- $1 : (\ \texttt{'a'} : [])$
- $1 :: \texttt{Integer}$, $[] :: [b]$, $\texttt{'a'} :: \texttt{Char}$ (Typen der Konstanten.)
- $(1 :) :: [\texttt{Integer}] \to [\texttt{Integer}]$ und $(\texttt{'a'} : [\,]) :: [\texttt{Char}]$.

Kein Typ als Resultat, denn:

$[\texttt{Integer}] \doteq [\texttt{Char}]$ ist nicht lösbar.

Beispiel zu Typfehler im Interpreter

```
Prelude> [1,'a'] ⏎

<interactive>:1:1:
    No instance for (Num Char)
      arising from the literal ‘1’ at <interactive>:1:1
    Possible fix: add an instance declaration for (Num Char)
    In the expression: 1
    In the expression: [1, 'a']
    In the definition of ‘it’: it = [1, 'a']
```

Typ eines Ausdrucks

Typ von `(map quadrat [1,2,3,4])` :

- `map`:: $(a \to b) \to [a] \to [b]$
- `quadrat`:: `Integer` $\to$ `Integer`, und $[1,2,3,4]$ :: $[$`Integer`$]$.
- $\gamma =$ $\{a \mapsto$ `Integer`$, b \mapsto$ `Integer`$\}$.
- Das ergibt: $\gamma(a) =$ `Integer`, $\gamma([a]) = [$`Integer`$]$, $\gamma([b]) = [$`Integer`$]$.
- Resultat: $\gamma([b]) = [$`Integer`$]$

Typisierung von Funktionen und Ausdrücken

Kompexe Sprachkonstrukte: rekursiv definierte Funktionen
Lambda-Ausdrücke, `let`-Ausdrücke
Listen-Komprehensionen.

Typcheckalgorithmus von Robin Milner (in Haskell und ML)

- ist schnell, (i.a.)
- liefert allgemeinste (Milner-)Typen
- benutzt Unifikation (eine optimierte Variante)

- schlechte worst-case Komplexität:
  in seltenen Fällen exponentieller Zeitbedarf
- Liefert in seltenen Fällen nicht
  die allgemeinsten möglichen *polymorphen* Typen

Typisierung von Funktionen, Milner

Satz zur Milner-Typisierung in Haskell:

Sei t ein getypter Haskell-Ausdruck, ohne freie Variablen.
Dann wird die Auswertung des Ausdrucks t
nicht mit einem Typfehler abbrechen.

Allgemeine Aussage zu starken Typsystemen

> Es gibt keine dynamischen Typfehler,
> wenn das Programm statisch korrekt getypt ist

Typisierung und Reduktion

Beachte: Nach Reduktionen kann ein Ausdruck
mehr Typen (bzw. einen allgemeineren Typ) haben
als vor der Reduktion

Beispiel:

```
if 1 > 0 then [] else [1]      :: [Integer]
```

arithmetische-Reduktion:

```
⟶ if True then [] else [1]     :: [Integer]
```

Case-Reduktion:

```
⟶ []                           :: [a]
```

Typisierung und Reduktion

weiteres Beispiel:

```
(if 1 > 0 then foldr else foldl) ::
       (a -> a -> a) -> a -> [a] -> a
```

Reduktion ergibt als Resultat:

```
foldr:: (a -> b -> b) -> b -> [a] -> b
```

der Typ ist allgemeiner geworden!