Praktische Informatik 3: Funktionale Programmierung
Vorlesung 7 vom 25.11.2014: Typinferenz

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Rekursive Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
  - Typinferenz
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt der Vorlesung

- Wozu Typen?
- Was ist ein Typsystem?
- Herleitung von Typen und Prüfung der Typkorrektheit (Typinferenz)

Wozu Typen?

- Frühzeitiges Aufdecken "offensichtlicher" Fehler
- "Once it type checks, it usually works"
- Hilfestellung bei Änderungen von Programmen
- Strukturierung großer Systeme auf Modul- bzw. Klassenebene
- Effizienz

Was ist ein Typsystem?

Ein Typsystem ist eine handhabbare syntaktische Methode, um die Abwesenheit bestimmter Programmverhalten zu beweisen, indem Ausdrücke nach der Art der Werte, die sie berechnen, klassifiziert werden.
(Benjamin C. Pierce, Types and Programming Languages, 2002)

Slogan:

Well-typed programs can't go wrong
(Robin Milner)

Typen in Haskell (The Story So Far)

- Primitive Basisdatentypen: Bool, Double
- Funktionstypen: Int → Int → Int, [Double] → Double
- Typkonstruktoren: [], (...), Foo
- Typvariablen:
  fst :: (α, β) → α
  length :: [α] → Int
  map :: (α → β) → [α] → [β]
- Typklassen :
  elem :: Eq a⇒ a → [a] → Bool
  max :: Ord a⇒ a → a → a

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

```
f m xs = m + length xs
```

- Frage: welchen Typ hat f?
  - Unterfrage: ist die angegebene Typsignatur korrekt?
- Informelle Ableitung

```
f m xs  =  m   +   length   xs
           Int     [α]→ Int
                        [α]
                    Int
               Int
          Int
f :: Int→ [α]→ Int
```


Typinferenz

- Mathematisch exaktes System zur Typbestimmung
  - Antwort auf die Frage: welchen Typ hat ein Ausdruck?
- Formalismus: Typableitungen der Form

$$\Gamma \vdash e :: \tau$$

  - Γ — Typumgebung (Zuordnung Symbole zu Typen)
  - e — Term
  - τ — Typ
- Beschränkung auf eine Kernsprache

Kernsprache: Ausdrücke

- Beschränkung auf eine kompakte Kernsprache:

$$\begin{array}{rcl} e & ::= & var \\ & | & \lambda\ var.\ e_1 \\ & | & e_1\ e_2 \\ & | & \textbf{let}\ var = e_1\ \textbf{in}\ e_2 \\ & | & \textbf{case}\ e_1\ \textbf{of} \\ & & \quad C_1\ var_1 \cdots var_n \to e_1 \\ & & \quad \ldots \end{array}$$

- Rest von Haskell hierin ausdrückbar:
  - **if**... **then**... **else**, Guards, Mehrfachapplikation, Funktionsdefinition, **where**

Kernsprache: Typen

- Typen sind gegeben durch:

$$\begin{array}{rcl} T & ::= & tvar \\ & | & C\ T_1\ \ldots\ T_n \end{array}$$

- *tvar* sind Typvariablen $\alpha, \beta, \ldots$
- C ist Typkonstruktur der Arität n. Beispiele:
  - Basistypen $n = 0$ (`Int`, `Bool`)
  - Listen $[t_1]$ mit $n = 1$
  - Funktionstypen $T_1 \to T_2$ mit $n = 2$

Typinferenzregeln

$$\frac{x :: t \in \Gamma}{\Gamma \vdash x :: t}\ Var \qquad \frac{\Gamma, x :: s \vdash e :: t}{\Gamma \vdash \lambda x.\ e :: s \to t}\ Abs$$

$$\frac{\Gamma \vdash e :: s \to t \quad \Gamma \vdash e' :: s}{\Gamma \vdash e\ e' :: t}\ App$$

$$\frac{\Gamma, x :: t_1 \vdash e_1 :: t_1 \quad \Gamma, x :: t_1 \vdash e_2 :: t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: t_2}\ LetRec$$

$$\frac{\Gamma \vdash f :: s \quad \Gamma, y_{i,j} :: t_{i,j} \vdash p_i :: s \quad \Gamma, y_{i,j} :: t_{i,j} \vdash e_i :: t}{\Gamma \vdash \textbf{case}\ f\ \textbf{of}\ p_i\ \to\ e_i :: t}\ Cases$$

Beispielableitung formal

- Haskell-Program: `f m xs = m + len xs`
- In unserer Sprache: $\lambda m\, xs.\ m + len\ xs$
- Initialer Kontext $C_0 \stackrel{def}{=} \{+ :: Int \to Int \to Int, len :: [\alpha] \to Int\}$
- Typableitungsproblem: $C_0 \vdash \lambda m\, xs.\ m + len\ xs :: ?$
- Ableitung als Baum:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{C_1 \vdash + :: Int \to Int \to Int}\ Var \quad \dfrac{}{C_1 \vdash m :: Int}\ Var}{C_1 \vdash m + :: Int \to Int}\ App \quad \dfrac{\dfrac{}{C_1 \vdash len :: [\alpha] \to Int}\ Var \quad \dfrac{}{C_1 \vdash xs :: [\alpha]}\ Var}{C_1 \vdash len\ xs :: Int}\ App}{C_1 \stackrel{def}{=} C_0, m :: Int, xs :: [\alpha] \vdash m + len\ xs :: Int}\ Abs}{C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int}\ Abs}{C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int}\ Abs$$

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

1.	$C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int$	*Abs*[2]
2.	$C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int$	*Abs*[3]
3.	$C_1 \stackrel{def}{=} C_0, m :: Int, xs :: [\alpha] \vdash m + len\ xs :: Int$	*App*[4, 7]
4.	$C_1 \vdash m + :: Int \to Int$	*App*[5, 6]
5.	$C_1 \vdash + :: Int \to Int \to Int$	*Var*
6.	$C_1 \vdash m :: Int$	*Var*
7.	$C_1 \vdash len\ xs :: Int$	*App*[8, 9]
8.	$C_1 \vdash len :: [\alpha] \to Int$	*Var*
9.	$C_1 \vdash xs :: [\alpha]$	*Var*

Problem: Typvariablen

- Sei $D \stackrel{def}{=} \{id : \alpha \to \alpha\}$
- Typableitung $D \vdash id\ id :: \alpha \to \alpha$ benötigt zwei unterschiedliche Instantiierung von *id*
- Andererseits: in $C \vdash \lambda x.\, x\ x :: ?$ darf x nicht unterschiedlich instantiiert werden.
- Deshalb: Typschemata

$$\begin{array}{rcl} S & ::= & \forall tvar.\ S \mid T \end{array}$$

- Zwei zusätzliche Regeln zur Instantiierung und Generalisierung

Typinferenzregeln (vollständig)

$$\frac{x :: t \in \Gamma}{\Gamma \vdash x :: t}\ Var \qquad \frac{\Gamma, x :: s \vdash e :: t}{\Gamma \vdash \lambda x.\ e :: s \to t}\ Abs$$

$$\frac{\Gamma \vdash e :: s \to t \quad \Gamma \vdash e' :: s}{\Gamma \vdash e\ e' :: t}\ App$$

$$\frac{\Gamma, x :: t_1 \vdash e_1 :: t_1 \quad \Gamma, x :: t_1 \vdash e_2 :: t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: t_2}\ LetRec$$

$$\frac{\Gamma \vdash f :: s \quad \Gamma, y_{i,j} :: t_{i,j} \vdash p_i :: s \quad \Gamma, y_{i,j} :: t_{i,j} \vdash e_i :: t}{\Gamma \vdash \textbf{case}\ f\ \textbf{of}\ p_i\ \to\ e_i :: t}\ Cases$$

$$\frac{\Gamma \vdash e :: \forall \alpha.\, t}{\Gamma \vdash e :: t\left[\frac{s}{\alpha}\right]}\ Spec \qquad \frac{\Gamma \vdash e :: t \quad \alpha \text{ nicht frei in } \Gamma}{\Gamma \vdash e :: \forall \alpha.\, t}\ Gen$$

Beispiel: *id* revisited

Damit ist jetzt $D \stackrel{def}{=} id : \forall \alpha.\, \alpha \to \alpha$

1.	$D \vdash id\ id :: \forall \beta.\, \beta \to \beta$	*Gen*
2.	$D \vdash id\ id :: \beta \to \beta$	*App*[3, 5]
3.	$D \vdash id :: (\beta \to \beta) \to (\beta \to \beta)$	*Spec*[4]
4.	$D \vdash id :: \forall \alpha.\, \alpha \to \alpha$	*Var*
5.	$D \vdash id :: \beta \to \beta$	*Spec*[6]
6.	$D \vdash id :: \forall \alpha.\, \alpha \to \alpha$	*Var*

Typinferenz: Typen ableiten

- Das Typinferenzproblem:
  - Gegeben Γ und e
  - Gesucht wird: τ und σ so dass $\sigma(\Gamma) \vdash e :: \tau$
- Berechung von τ und σ durch Algorithmus W (Damas-Milner)
- Informell:
  - Typbestimmung beginnt an den Blättern des Ableitungsbaumes (Regeln ohne Voraussetzungen)
  - Konstanten zuerst (Typ fest), dann Variablen (Typ offen)
  - Instantiierung für Typschemata und Typ für Variablen unbestimmt lassen, und konsistent anpassen
  - Typ mit Regeln wie Abs, App und Cases nach oben propagieren

Beispiel: Bestimmung des Typs von map

Die Funktion map:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

In unserer Kernsprache:

let $map = \lambda f\ ys.$ **case** ys *of* $[] \to []; (x : xs) \to map\ f\ y$ **in** map

Ableitung mit $C_0 \stackrel{def}{=} \{(:) : \forall\alpha.\alpha \to [\alpha] \to \alpha\}$

Beispiel: Bestimmung eines großen Typen

Die Sequenzierungsoperation für Parser:

```
(>*>) p1 p2 i =
  concatMap (λ(b, r)→
     map (λ(c, s)→ ((b, c), s)) (p2 r)) (p1 i)
```

In unserer Kernsprache:

$\lambda p1\ p2\ i.\ concatMap(\lambda br.\ map(\lambda cs.\ ((fst\ br, fst\ cs), snd\ cs))(p2\ (snd\ br)))(p1 i)$

Eigenschaften von W

- Entscheidbarkeit und Korrektheit: Für Kontext Γ und Term e terminiert Algorithmus W immer, und liefert $W(\Gamma, e) = (\sigma, \tau) = (\sigma, \tau)$ so dass $\sigma(\Gamma) \vdash e :: \tau$
- Vollständigkeit: $W(\Gamma, e)$ berechnet den allgemeinsten Typen (principal type) von e (wenn es ihn gibt)
- Aufwand von W:
  - Theoretisch: exponentiell (*DEXPTIME*)
  - Praktisch: in relevanten Fällen annähernd linear

Typen in anderen Programmiersprachen

- Statische Typisierung (Typableitung während Übersetzung)
  - Haskell, ML
  - Java, C++, C (optional)
- Dynamische Typisierung (Typüberprüfung zur Laufzeit)
  - PHP, Python, Ruby (*duck typing*)
- Ungetypt
  - Lisp, LaTeX, Tcl, Shell

Zusammenfassung

- Haskell implementiert Typüberprüfung durch Typinferenz (nach Damas-Milner)
- Kernelemente der Typinferenz:
  - Bindung von Typvariablen in Typschema ($\forall\alpha.\tau$)
  - Berechnung des Typen von den Blättern des Ableitungsbaumes her
  - Typinferenz berechnet allgemeinsten Typ
- Typinferenz hat praktisch linearen, theoretisch exponentiellen Aufwand
- Nächste Woche: Module und abstrakte Datentypen in Haskell