Praktische Informatik 3: Funktionale Programmierung
Vorlesung 7 vom 27.11.2012: Typinferenz

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

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
         [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

$$\Gamma \vdash x :: t$$

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

Kernsprache: Ausdrücke

- Beschränkung auf eine kompakte Kernsprache:

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

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

Kernsprache: Typen

- Typen sind gegeben durch:

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

- *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 $n = 2$
- Typschemata sind gegeben durch:

$$S ::= \forall tvar.\ S \mid T$$

Typinferenzregeln

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

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

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

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

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

Typinferenz: Algorithmus W

$$\begin{array}{rcl} W(\Gamma, x) = & (Id, \tau) & x :: \tau \in \Gamma \\ W(\Gamma, x) = & (Id, \tau\left[\begin{smallmatrix} \beta \\ \alpha \end{smallmatrix}\right]) & x :: \forall \alpha.\ \tau \in \Gamma, \beta\ \textit{frisch} \\ W(\Gamma, e_1\ e_2) = & \textbf{let}\ (\sigma_1, \tau_1) = & W(\Gamma, e_1) \\ & \quad (\sigma_2, \tau_2) = & W(\sigma_1(\Gamma), e_2) \\ & \quad u = & \mathit{unify}(\sigma_2(\tau_1), \tau_2 \to \beta) \\ & & \beta\ \textit{frisch} \\ & \textbf{in}\ (u \cdot \sigma_2 \cdot \sigma_1, u(\beta)) & \\ W(\Gamma, \lambda x.\ e) = & \textbf{let}\ (\sigma, \tau) = W(\Gamma \uplus \{x :: \beta\}, e) & \beta\ \textit{frisch} \\ & \textbf{in}\ (\sigma, \sigma(\beta \to \tau)) & \\ W(\Gamma, \textbf{let}\ x = e_1\ \textbf{in}\ e_2) = & \textbf{let}\ (\sigma_1, \tau_1) = & W(\Gamma, e_1) \\ & \quad (\sigma_2, \tau_2) = & W(\sigma_1(\Gamma) \uplus \{x :: \overline{\Gamma}(\tau_1)\}, e_2) \\ & \textbf{in}\ (\sigma_2 \cdot \sigma_1, \tau_2) & \end{array}$$

Eigenschaften von W

- Korrektheit: Wenn $(\sigma, \tau) = W(\Gamma, e)$, dann $\sigma(\Gamma) \vdash e :: \tau$
- Vollständigkeit: $W(\Gamma, e)$ berechnet den allgemeinsten Typen (principal type) von e (wenn es ihn gibt)
- Aufwand von W:
  - Theoretisch: exponentiell ($DTIME(2^{n^{O(1)}})$)
  - Praktisch: in relevanten Fällen annähernd linear

Substitution und Unifikation

- Substitution: $t\left[\begin{smallmatrix} s \\ a \end{smallmatrix}\right]$ ersetzt Variable a in t durch s
- Substitutionen können komponiert werden
- Substitutition σ_1 ist allgemeiner als σ_2, wenn $\sigma_2 = \tau \cdot \sigma_1$
- Unifikator zweier Terme s, t: Substitution u so dass $us = ut$
- Unifikationsalgorithmus nach Robinson berechnet allgemeinsten Unifikator

Unifikationsalgorithmus nach Robinson

$$\begin{array}{lll} \mathit{unify}(s, t) & & \\ s \equiv \alpha & \mid t \equiv \alpha & = Id \\ & \mid \mathit{occurs}(\alpha, t) & = error \quad \text{— Siehe (1) unten} \\ & \mid otherwise & = \{\alpha \mapsto t\} \\ t \equiv \alpha & \mid \mathit{occurs}(\alpha, s) & = error \quad \text{— Siehe (1) unten} \\ & \mid otherwise & = \{\alpha \mapsto s\} \\ s \equiv C\ s_1 \ldots s_n, t \equiv D\ t_1 \ldots t_m & & \\ & \mid C \neq D & = error \quad \text{— Siehe (2) unten} \\ & \mid otherwise & = foldl\ \ (\lambda m\ (s_i, t_i).\ \mathit{unify}(m\ s_i, m\ t_i)) \\ & & \quad Id \\ & & \quad (zip\ [s_1, \ldots, s_n]\ [t_1, \ldots t_m]) \\ & & \text{— Rekursive Unifikation der Subterme } s_i,\ t_i \end{array}$$

Der Algorithmus schlägt fehl wenn

1. eine Typariable α mit einem Term t ersetzt werden soll, der α enthält
2. zwei unterschiedliche Typkonstruktoren unifiziert werden sollen

Beispiele

- Typinferenz (nach W) für

$$\lambda xs.\ tail(head xs)$$

- Typinferenz (pragmatisch) für

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

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:
  - Typunifikation (*unify*) bei Applikation
  - Bindung von Typvariablen in Typschema ($\forall\alpha.\tau$)
  - Typinferenz berechnet allgemeinsten Typ
- Typinferenz hat annähernd linearen Aufwand (kann exponentiell werden)