Fortgeschrittene Funktionale Programmierung

12. Vorlesung

Janis Voigtländer

Universität Bonn

Wintersemester 2015/16

Motivation

Auf `http://www-ps.iai.uni-bonn.de/ft`:

This tool allows to generate free theorems for sublanguages of Haskell as described here.

The source code of the underlying library and a shell-based application using it is available here and here.

Please enter a (polymorphic) type, e.g. "(a -> Bool) -> [a] -> [a]" or simply "filter":

```
g :: (a -> Bool) -> [a] -> [a]
```

Please choose a sublanguage of Haskell:

- (•) no bottoms (hence no general recursion and no selective strictness)
- ( ) general recursion but no selective strictness
- ( ) general recursion and selective strictness

Please choose a theorem style (without effect in the sublanguage with no bottoms):

- (•) equational
- ( ) inequational

[Generate]

Motivation

The theorem generated for functions of the type

```
g :: forall a . (a -> Bool) -> [a] -> [a]
```

in the sublanguage of Haskell with no bottoms is:

```
forall t1,t2 in TYPES, R in REL(t1,t2).
 forall p :: t1 -> Bool.
  forall q :: t2 -> Bool.
   (forall (x, y) in R. p x = q y)
   ==> (forall (z, v) in lift{[]}(R).
         (g p z, g q v) in lift{[]}(R))
```

The structural lifting occurring therein is defined as follows:

```
lift{[]}(R)
  = {([], [])}
  u {(x : xs, y : ys) |
       ((x, y) in R) && ((xs, ys) in lift{[]}(R))}
```

Reducing all permissible relation variables to functions yields:

```
forall t1,t2 in TYPES, f :: t1 -> t2.
 forall p :: t1 -> Bool.
  forall q :: t2 -> Bool.
   (forall x :: t1. p x = q (f x))
   ==> (forall y :: [t1]. map f (g p y) = g q (map f y))
```

Export as PDF Show type instantiations Enter a new type Help page

Betrachtungen zu Polymorphie

Charakterisierung von Funktionen $g :: [a] \to [a]$:

- Wir behaupten, dass für alle $g_1, g_2 :: [a] \to [a]$ gilt:

  $g_1 \equiv g_2$ wenn für alle $xs :: [\mathsf{Int}]$, $g_1\ xs \equiv g_2\ xs$

  Können wir das denn auch beweisen?

- Wie sieht es denn aus mit, für alle $g_1, g_2 :: [a] \to [a]$:

  $g_1 \equiv g_2$ wenn für alle $xs :: [\mathsf{Bool}]$, $g_1\ xs \equiv g_2\ xs$?

- Und/oder können wir auf die Quantifizierung von xs über alle Listen des entsprechenden Typs verzichten?

- Was für Antworten ergeben sich, wenn wir von $[a] \to [a]$ zu $\mathsf{Eq}\ a \Rightarrow [a] \to [a]$ übergehen?

- ...

Programming Language Meta-Theory

Wir wollen eine formale Sprachbeschreibung (mit Details zu Syntax, Typsystem, Semantik) für Haskell, bzw. für eine ausgewählte Teilsprache davon.

Einige Beschränkungen, die wir zum Zwecke der Beherrschbarkeit in Kauf nehmen werden:

- stark vereinfachte Syntax, ohne einen Großteil von Haskells syntactic sugar
- nur ein Ausschnitt aus der Welt der Haskell-Datentypen
- nicht alle Formen von Polymorphie, keine Typklassen
- alles als Term, keine gesonderten Programmgleichungen

Ein einfacher Lambda-Kalkül: (vereinfachte) Syntax

Termgrammatik: $t := x \mid \mathsf{n} \mid \lambda x.t \mid t\ t \mid t + t \mid$
$\textbf{if}\ t == 0\ \textbf{then}\ t\ \textbf{else}\ t$

wobei:

- n (0, 1, 2, ...): konkrete Zahlen-Literale
- x (und y, z, ...): Termvariablen (in Termen vorkommende ...)
- (t, u als Metavariablen *für* Terme)
- zunächst ungetypte Funktionsabstraktion und -applikation
- relativ willkürliche Operationen zum Kombinieren und Konsumieren numerischer Werte

Beispielterme:

- $\lambda x.(\textbf{if}\ x == 0\ \textbf{then}\ 3\ \textbf{else}\ x + 1)$
- $(\lambda x.(\textbf{if}\ x == 0\ \textbf{then}\ 3\ \textbf{else}\ x + 1))\ 2$
- $\lambda x.\lambda y.(x\ y) + 4$

Begrifflichkeiten:

- Scope von Variablen, freie und gebundene Vorkommen

Ein einfacher Lambda-Kalkül: (vereinfachte) Syntax

Was meinte „alles als Term, keine gesonderten Programmgleich.“?

Dass wir keine Programme wie etwa

$$\texttt{f} = \lambda x.(\textbf{if } x == 0 \textbf{ then } 3 \textbf{ else } x + 1)$$
$$\texttt{k} = \texttt{f}\ 2$$
$$\texttt{g}\ x = \lambda y.(x\ y) + 4$$
$$\texttt{main} = \texttt{g}\ \texttt{f}\ \texttt{k}$$

haben werden, sondern immer genau einen zu betrachtenden Gesamtterm, ohne weiteren Programmkontext:

$$(\lambda x.\lambda y.(x\ y) + 4)\ (\lambda x.(\textbf{if } x == 0 \textbf{ then } 3 \textbf{ else } x + 1))\ ((\lambda x.(\textbf{if } x == 0 \textbf{ then } 3 \textbf{ else } x + 1))\ 2)$$

Potentielle Probleme mit dieser Betrachtung:

- aus pragmatischer Sicht: Modularität, separate compilation?
- aus theoretischer Sicht: Was ist mit Rekursion?
- falls mal Effizienz von Interesse: kein Top-Level-Sharing?

Ein einfacher Lambda-Kalkül: Typsystem

- Um nur „sinnvolle“ Terme zu betrachten, Einführung von Typen.
- Für die bisher gesehenen Beispiele würden aus Nat und $\rightarrow$ aufgebaute Typen reichen.
- Wir führen aber gleich auch noch Typvariablen (α, β, ...) ein (für letztlich Polymorphie), also:

  Typgrammatik: $\tau := \alpha \mid \tau \rightarrow \tau \mid \text{Nat}$
- Auf Termebene führen wir lediglich bei Funktionsabstraktionen eine Typannotation ein: $\lambda x : \tau . t$ statt $\lambda x . t$
- Schließlich brauchen wir „nur“ noch eine Beschreibung, welche Terme wohlgetypt sind, und mit welchen konkreten Gesamttypen ...

Ein einfacher Lambda-Kalkül: Typsystem

Typen: $\tau := \alpha \mid \tau \to \tau \mid \mathsf{Nat}$

Terme: $t := x \mid \mathsf{n} \mid \lambda x : \tau . t \mid t\ t \mid t{+}t \mid \textbf{if } t == 0 \textbf{ then } t \textbf{ else } t$

$$\Gamma, x : \tau \vdash x : \tau$$

$$\Gamma \vdash \mathsf{n} : \mathsf{Nat}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . t) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t : \mathsf{Nat} \qquad \Gamma \vdash u : \mathsf{Nat}}{\Gamma \vdash (t + u) : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash t : \mathsf{Nat} \qquad \Gamma \vdash u_1 : \tau \qquad \Gamma \vdash u_2 : \tau}{\Gamma \vdash (\textbf{if } t == 0 \textbf{ then } u_1 \textbf{ else } u_2) : \tau}$$

Begrifflichkeiten:

- typing contexts Γ
- typing judgements
- well-formedness
- freshness conventions
- type derivations

Beispiel für eine Typableitung

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, y : \gamma \vdash f : \alpha \to \beta \qquad \Gamma, y : \gamma \vdash x : \alpha}{\Gamma, y : \gamma \vdash (f\ x) : \beta}}{\Gamma \vdash (\lambda y : \gamma . f\ x) : \gamma \to \beta} \qquad \Gamma \vdash g : (\gamma \to \beta) \to \gamma}{\Gamma \vdash (g\ (\lambda y : \gamma . f\ x)) : \gamma}}{\alpha, \beta, \gamma, f : \alpha \to \beta, g : (\gamma \to \beta) \to \gamma \vdash (\lambda x : \alpha . g\ (\lambda y : \gamma . f\ x)) : \alpha \to \gamma}}{\alpha, \beta, \gamma, f : \alpha \to \beta \vdash (\lambda g : \cdots . \lambda x : \alpha . g\ (\lambda y : \gamma . f\ x)) : ((\gamma \to \beta) \to \gamma) \to \alpha \to \gamma}}{\alpha, \beta, \gamma \vdash (\lambda f : \alpha \to \beta . \lambda g : (\gamma \to \beta) \to \gamma . \lambda x : \alpha . g\ (\lambda y : \gamma . f\ x)) : \tau}$$

wobei $\tau = (\alpha \to \beta) \to ((\gamma \to \beta) \to \gamma) \to \alpha \to \gamma$

$\Gamma = \alpha, \beta, \gamma, f : \alpha \to \beta, g : (\gamma \to \beta) \to \gamma, x : \alpha$