Typklassen

Janosch Kindl

16. Dezember 2016

Zusammenfassung

Dies ist die Ausarbeitung zum gleichnamigen Seminarvortrag aus "Perlen der Informatik 3". Es wird zuerst der Begriff Ad-hoc Polymorphismus anhand von bereits bekannten Porgrammiersprachen erklärt. Dann geht es weiter zu Typklassen in Haskell und wie der Compiler polymorphe Ausdrücke in die Verwendung von Funktionstabellen übersetzt. Am Ende sehen wir die Inferenzregeln von Wadler und Blott aus dem Jahre 1988, die die Typisierung und Übersetzung für eine vereinfachte Kernsprache formalisieren.

1 Einleitung

Bei der Entwicklung der Programmiersprache Haskell wurde es sich zur Aufgabe gemacht, etablierte Lösungen zu verwenden. Man musste aber festellen, dass es für Funktionsüberladung noch keine etablierte Lösung gab. *Ad-hoc* ist aus dem Lateinischen und bedeutet wörtlich *hierfür*. Auch wenn Ad-hoc für spontane und improvisierte Lösungen steht, sind Typklassen eine elegante Antwort auf die Frage nach Funktionsüberladung und sie sind mitbeteiligt am Erfolg von Haskell. Wadler und Blott nannten ihre Veröffentlichung, auf der diese Ausarbeitung vorwiegend beruht, treffend "How to make ad-hoc polymorphism less ad-hoc" [2].

Doch was bedeutet Ad-hoc Polymorphismus? Aus der Vorlesung Informatik 2 kennen wir bereits parametrisch polymorphe Funktionen. Beim parametrischen Polymorphismus ist eine Funktion auf mehreren Typen definiert und verhält sich für alle diese Typen gleich. Ein Beispiel ist die Funktion *length* die die Länge einer Liste zurückgibt und sowohl für Listen vom Typ Integer als auch vom Typ Float funktioniert. Beim Ad-hoc Polymorphismus verhält sich eine Funktion für jeden Typ anders. Beispiel hierfür wäre eine Funktion *toString* die einen Wert in seine textuelle Darstellung umwandelt.

In der Typsystemreihe des Seminars wurde bereits der untypisierte und der einfach typisierte Lambda-Kalkül vorgestellt. Danach haben wir gesehen, wie man mit den Regeln von Damas und Milner parametrisch polymorphe Funktionen typisieren kann. In dieser Ausarbeitung werden wir die Regeln von Wadler und Blott kennenlernen, die sie um Funktionsüberladung erweitern und während der Typisierung einen Ausdruck, der überladene Funktionen enthält, in einen Ausdruck übersetzen, der frei davon ist.

2 Ad-hoc Polymorphismus

Aus Sprachen wie Java kennen wir überladene Operatoren, wie etwa die arithmetischen Operatoren *+*, *-*, *** und */*. Diese sind sowohl auf Float- als auch auf Integerwerte anwendbar. In OCaml gibt es für Integer und Floats einen eigenen Operator. Beispielsweise für Integer den Operator *(+)* und für Floats den Operator *(+.)*. Nachteil dieser Eindeutigkeit ist, dass wenn man eine Funktion *sum* definieren wollte, die eine Liste von Zahlen addiert, man eine Implementierung für Listen von Integern als auch für Floats angeben müsste:

```
let rec sumi = function
  [] -> 0
  | x::xs -> x + sumi xs;;

let rec sumf = function
  [] -> 0
  | x::xs -> x +. sumf xs;;
```

Man erkennt sofort, dass die Implementierungen fast identisch sind. Es wäre schön, wenn man die Funktion präziser und allgemeiner ausdrücken könnte. Es gibt in OCaml aber auch den *(=)* Operator, der sich anders verhält. Er ist vom Typ:

```
(=) : 'a -> 'a -> bool
```

Dieser Operator ist polymorph und sein Typ lässt den Eindruck entstehen, dass sich alle Werte vergleichen liesen, wenn sie vom selben Typ sind. Wenn wir jetzt aber den Ausdruck

```
(fun x -> x) = (fun x -> x);;
```

auswerten, dann wird die Ausnahme *Invalid_argument "compare: functional value"* zur Ausführung des Programms geworfen. Es wäre besser, wenn das Typsystem solche Feler bereits während des kompilierens finden könnte. In Standard ML gibt es *equality types*, Typvariablen die nur mit Typen unifiziert werden, auf denen strukturelle Gleichheit definiert ist. Dies schließt den Vergleich von Funktionstypen bereits zur Kompilierungszeit aus, aber man kann selbst keine überladene Operatoren oder eigene Typvariablen im selben Stil definieren.

Bevor wir weiter zu Typklassen in Haskell schreiten, gehen wir einen Schritt zurück und schauen uns objektorientiere Sprachen an. Hier sehen wir eine Klasse *IntTupel*, mit zwei Integerwerten und einer Definition der Methode *equals*.

```
class IntTupel extends Object {
  int x; int y;
  public boolean equals(Object o) {
    if(!(o instanceof IntTupel))
      return false;
    IntTupel that = (IntTupel)o;
    return this.x == that.x && this.y == that.y;
```

```
  }
}
```

Diese Methode ist bereits in der Superklasse *Object* definiert, von der *IntTupel* erbt. Die Definition überschreibt die Methode mit einer spezifischeren Methode der Klasse *IntTupel*. Hiermit können wir schon simple polymorphe Programme schreiben wie etwa:

```
Object a = new IntTupel();
IntTupel b = new IntTupel();
a.equals(b);
```

Der statische Typ von *a* ist *Object*, dennoch wird zur Laufzeit anhand der Funktiontabelle des ersten Objekts, die spezifische *equals* Methode der Klasse *IntTupel* bestimmt und aufgerufen. Dieser Vorgang nennt sich *Single Dispatch* und diese Art des Polymophismus nennt sich *Subtyping Polymorphism*, da er auf der Vererbung von Klassen beruht. In diesem Fall haben aber beide Objekte eine Funktionstabelle. Die Idee bei Typklassen ist, diese Funktiontabelle unabhängig von den Objekten als eigener Parameter zu übergeben.

3 Typklassen in Haskell

In Haskell wurden die Probleme, die wir in OCaml und Standard ML gesehen haben mit Typklassen gelöst. Am folgendem Beispiel lässt sich am besten zeigen, wie Gleichheit in Haskell definiert werden könnte:

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int where
  -- eine eingebaute Funktion
  -- vom Typ Int -> Int -> Bool
  (==) = eqInt

instance Eq a, Eq b => Eq (a, b) where
  (u, v) == (x, y) =
    (u == x) && (v == y)
```

Wir sehen eine class-Deklaration in der wir festlegen, dass es eine Klasse Eq für den Typ a gibt, die die Funktion *(==)* besitzt. Wir verwenden den Operator *(==)* mit zwei Gleichheitszeichen, damit er syntaktisch von Definitionen unterschieden werden kann. In der instance-Deklaration steckt die eigentliche Implementierung für einen spezifischen Typ. In dem Beispiel für Int verwenden wir eine theoretische, eingebaute Funktion, die zwei Integer miteinander vergleicht. Man ist aber nicht auf eingebaute Funkitonen beschränkt, wie das Beispiel für Tupel zeigt. Hier vergleichen wir jeweils die ersten Elemente des Tupels und die zweiten Elemente des Tupels miteinander. Interessant ist die erste Zeile dieser Instanz:

```
instance Eq a, Eq b => Eq (a, b)
```

Wir verwenden zum Vergleich der Elemente des Tupels wieder den (==) Operator, deswegen muss er für die Typen im Tupel auch definiert sein. Das beinhaltet diese instance-Deklaration. Wenn man beispielsweise zwei Tupel des Typs (Int, Char) vergleichen möchte, muss es also eine Instanz Eq Int als auch Eq Char geben.

Der Typ des (==) Operators ist jetzt:

```
(==) :: Eq a => a -> a -> Bool
```

Dies ist ein Prädikatstyp, der besagt, dass es für den Typ a eine Instanz Eq geben muss. Prädikatstypen verallgemeinern das Konzept der *equality types* aus Standard ML und Voraussetzungen wie *Eq a* werden in der englischen Literatur häufig als *Constraint* bezeichnet.

Der Operator kann jetzt wie jede andere Funktion verwendet werden und es werden sogar Prädikatstypen inferiert.

```
-- inferierter Typ: Eq a => a -> a -> Bool
a /= b = not (a == b)
```

Eine Klasse kann natürlich auch mehrere Funktionen enthalten. Ein weiteres Beispiel für eine häufig Verwendete Typklasse ist *Num a*, die arithmetische Operatoren beinhaltet:

```
class  Num a  where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

Damit können wir jetzt eine Funktion *sum* definieren, die sowohl auf Listen von Floats als auch Integern anwendbar ist:

```
sum :: Num t => [t] -> t
sum [] = 0
sum (x:xs) = x + sum xs
```

3.1 Übersetzung

In den vorigen Beispielen haben wir die Verwendung des *(==)* Operators gesehen, aber kommen wir zurück zu der eigentlichen Idee hinter Typklassen, nämlich dass Funktionstabellen unabhängig vom Objekt, als eigener Parameter übergeben werden können. Der Compiler übersetzt die Anwendung des Operators in die Verwendung von Funktionstabellen (im Englischen *dictionary* genannt), wie wir hier sehen können:

```
-- class Eq a
data EqD a = EqDict (a -> a -> Bool)
-- "entpackt" die Funktion
eq (EqDict e) = e

-- instance Eq Int
eqDInt :: EqD Int
eqDInt = EqDict eqInt

-- instance Eq a, Eq b => Eq (a, b)
eqDPair :: (EqD a, EqD b) -> EqD (a, b)
eqDPair (eqDa, eqDb) =
  EqDict (eqPair (eqDa, eqDb))

eqPair (eqDa, eqDb) (x, y) (u, v) =
  eq eqDa x u && eq eqDb y v
```

Aus einer class-Deklaration wird ein Datentyp, mit einem Konstruktor, der eine Funktion enthält. Außerdem wird eine Funktion generiert, die diese Funktion aus dem Konstruktor wieder entpackt. Bei Typklassen mit mehreren Funktionen, wie etwa Num, würde der Konstruktor einfach mehrere Funktionen als Parameter besitzen.

3.2 Subklassen

Denkt man an folgende Klasse und Funktionsdefinition:

```
class Ord a where
  (<=) :: a -> a -> Bool

a < b  =  (a <= b) && not (a == b)
```

dann ist der inferierte Typ:

```
(<) :: (Eq a, Ord a) => a -> a -> Bool
```

Es müssen dann gleich zwei Funktionstabellen übergeben werden und in manchen Fällen kann die Anzahl benötigter Funktionstabellen schnell wachsen. Haskell bietet zur besseren Struktur Subklassen an. Die class-Deklaration und der inferierte Typ sehen dann wie folgt aus:

```
class Eq a => Ord a where
  (<=) :: a -> a -> Bool

(<) :: Ord a => a -> a -> Bool
```

Ord ist jetzt eine Subklasse von Eq. Das Prädikat *Ord a* impliziert *Eq a*. Die Übersetzung von Subklassen ist auch intuitiv. Es wird einfach die Funktionstabelle der Superklasse in die Funktionstabelle der Subklasse gepackt [1]:

```
-- ... data EqD a = EqDict (a -> a -> Bool)
data OrdD a = OrdDict (EqD a) (a -> a -> Bool)

eqDictFromOrd (OrdDict x _) = x
lte (OrdDict _ x) = x

(<) ordDict a b =
  lte ordDict a b &&
    not (eq (eqDictFromOrd ordDict) a b)
```

Die Tabelle für Ord enthält jetzt die Tabelle von Eq. Mit Subklassen kann man Typklassen strukturieren und das Übergeben von Funktionstabellen einsparen, aber sie sind nicht zwingend notwendig.

4 Inferenzregeln

Die folgenden Inferenzregeln sind von Philip Wadler und Stephen Blott aus dem Jahre 1988. Diese Regeln sind nicht auf Haskell anwendbar sondern formalisieren eine vereinfachte Kernsprache. Zu erst schauen wir uns ihre Syntax an:

$$
\begin{aligned}
\textit{Expressions} \quad & e ::= x \mid e_0\ e_1 \mid \lambda x.\ e \\
& \mid \textit{let}\ x = e_0\ \textit{in}\ e_1 \\
& \mid \textit{over}\ x :: \sigma\ \textit{in}\ e \\
& \mid \textit{inst}\ x :: \sigma = e_0\ \textit{in}\ e_1 \\
\textit{Types} \quad & \tau ::= (\tau \to \tau') \mid \alpha \mid \chi(\tau_1 ... \tau_n) \\
\textit{Predicated Types} \quad & \rho ::= (x :: \tau).\ \rho \mid \tau \\
\textit{Type Schemes} \quad & \sigma ::= \forall \alpha.\ \sigma \mid \rho
\end{aligned}
$$

Bei den Ausdrücken wurde das gewohnte Lambda-Kalkül mit Let um die Ausdrücke *over* und *inst* erweitert, die class- und instance-Deklarationen widerspigeln. Bei den Typen sind Typkonstruktoren χ dazugekommen. Beispielsweise ist $Eq\ \alpha$ der Typkonstruktor Eq angewendet auf die Typvariable α. Es gibt jetzt auch Prädikatstypen ρ, die wir vorher in anderer Form gesehen haben. Hier ein Beispiel für einen Ausdruck in der Kernsprache:

$$
\begin{aligned}
& \textit{over}\ eq :: \forall \alpha.\ Eq\ \alpha\ \textit{in} \\
& \textit{inst}\ eq :: Eq\ Int = eqInt\ \textit{in} \\
& \textit{inst}\ eq :: Eq\ Char = eqChar\ \textit{in} \\
& \textit{inst}\ eq :: \forall \alpha. \forall \beta. (eq :: Eq\ \alpha). (eq :: Eq\ \beta). Eq\ (\alpha, \beta) \\
& \quad = \lambda p. \lambda q. eq\ (fst\ p)\ (fst\ q) \wedge eq\ (snd\ p)\ (snd\ q)\ \textit{in} \\
& \qquad eq\ (1,\ 'a')\ (2,\ 'b')
\end{aligned}
$$

Es wird ein überladener Bezeichner *eq* eingeführt und es werden konkrete Instanzen für die Typen Int, Char und für Tupel angegeben. Danach steht ein Ausdruck, der zwei Tupel vom Typ (Int, Char) miteinander vergleicht.

Die Regeln sind von der Form $A \vdash e :: \sigma \setminus \bar{e}$ und lesen sich als "Unter der Menge der Annahmen A hat der Ausdruck e den Typ σ mit der Übersetzung $\bar{e}$." Rein für die Typisierung könnte man den Teil mit der Übersetzung auch weglassen, aber die Übersetzung kann nicht unabhängig von der Typisierung geschehen. Die Annahmen haben die Form $(x ::_o \sigma)$ für überladene Bezeichner, $(x ::_i \sigma \setminus x_\sigma)$ für die Instanzen der überladenen Bezeichner und $(x :: \sigma \setminus \bar{x})$ für Bezeichner die über Let- und Lambdaausdrücke gebunden sind.

Der folgende Abschnitt listet die Regeln auf und erklärt knapp die Intuition hinter den einzelnen Regeln. Die ersten Regeln sind denen von Hindley und Milner sehr ähnlich, nur die Übersetzung ist dazugekommen.

- TAUT $\dfrac{}{A, (x :: \sigma \setminus \bar{x}) \vdash x :: \sigma \setminus \bar{x}}$
- TAUT $\dfrac{}{A, (x ::_i \sigma \setminus \bar{x}) \vdash x :: \sigma \setminus \bar{x}}$

Bei den Regeln TAUT ermitteln wir den Typ eines Bezeichners im Kontext. Hier werden zwei Regeln benötigt, da es jetzt unterschiedliche Formen von Annahmen gibt.

- COMB $\dfrac{A \vdash e :: (\tau' \to \tau) \setminus \bar{e} \quad A \vdash e' :: \tau' \setminus \bar{e}'}{A \vdash (e\ e') :: \tau \setminus (\bar{e}\ \bar{e}')}$
- ABS $\dfrac{A_x, (x :: \tau' \setminus x) \vdash e :: \tau \setminus \bar{e}}{A \vdash (\lambda x.\ e) :: (\tau' \to \tau) \setminus (\lambda x.\ \bar{e})}$
- LET $\dfrac{A \vdash e :: \sigma \setminus \bar{e} \quad A_x, (x :: \sigma \setminus x) \vdash e' :: \tau \setminus \bar{e}'}{A \vdash (let\ x = e\ in\ e') :: \tau \setminus (let\ x = \bar{e}\ in\ \bar{e}')}$
- GEN $\dfrac{A \vdash e :: \sigma \setminus \bar{e} \quad \alpha\ not\ free\ in\ A}{A \vdash e :: \forall \alpha.\ \sigma \setminus \bar{e}}$
- SPEC $\dfrac{A \vdash e :: \forall \alpha.\ \sigma \setminus \bar{e}}{A \vdash e :: [\alpha \backslash \tau]\sigma \setminus \bar{e}}$

Die COMB Regel beschreibt die Funktionsanwendung. Der Ausdruck e muss einen Funktionstyp haben und der Parameter e' muss einen kompatiblen Typ besitzen. Die Regel ABS ist für Lambda-Ausdrücke und LET für Let-Ausdrücke. Es wird jeweils der Bezeichner x mit einem Typ den Annahmen hinzugefügt. GEN führt gebundene Typvariablen ein und SPEC eleminiert sie.

- PRED $\dfrac{A, (x :: \tau \setminus x_\tau) \vdash e :: \rho \setminus \bar{e}}{A \vdash e :: (x :: \tau).\ \rho \setminus (\lambda x_\tau.\ \bar{e})}\ (x ::_o \sigma) \in A$
- REL $\dfrac{A \vdash e :: (x :: \tau).\ \rho \setminus \bar{e} \quad A \vdash x :: \tau \setminus \bar{e}'}{A \vdash e :: \rho \setminus (\bar{e}\ \bar{e}')}\ (x ::_o \sigma) \in A$

Die Regeln PRED und REL sind analog zu GEN und SPEC. Ihre Namen kommen von *predicate* und *release*. Interessant ist vorallem die Wirkung der Regeln auf die Übersetzung. PRED beschreibt, dass ein Prädikatstyp in einen zusätzlichen

Parameter umgewandelt wird, nämlich der Funktionstabelle. Die Regel REL übersetzt die Anwendung eines Überladenen Bezeichners in die Extraktion einer spezifischen Funktion aus der Funktionstabelle.

- OVER $\dfrac{A_x, (x ::_o \sigma) \vdash e :: \tau \setminus \bar{e}}{A \vdash (over\ x :: \sigma\ in\ e) :: \tau \setminus \bar{e}}$
- INST $\dfrac{A, (x ::_i \sigma' \setminus x_{\sigma'}) \vdash e' :: \sigma' \backslash \bar{e}' \quad A, (x ::_i \sigma' \setminus x_{\sigma'}) \vdash e :: \tau \backslash \bar{e}}{A \vdash (inst\ x :: \sigma' = e' in\ e) :: \tau \setminus (let\ x_{\sigma'} = \bar{e}' in\ \bar{e})}$

Diese Regeln sind für die neuen syntaktischen Konstrukte. Bei OVER wird ein überladener Bezeichner den Annahmen hinzugefügt und der Ausdruck verschwindet in der Übersetzung einfach. Bei INST wird eine Instanz des Bezeichners den Annahmen hinzugefügt und der Ausdruck wird in einen Let-Ausdruck übersetzt. Nach der Übersetzung enthält der Ausdruck also keine over- und inst-Ausdrücke mehr.

Wir haben Operatoren in der Sprache OCaml und Subtyping Polymorphism in der Sprache Java gesehen, die uns langsam an Typklassen in Haskell herangeführt haben. Es wurde informell die Übersetzung von Typklassen in Haskell und danach die formalen Regeln auf einer einfacheren Kernsprache gezeigt. Heute gibt es viele andere Formalisierungen von Typklassen, aber diese Regeln eignen sich besser zum Erklären der Konzepte, da sie den Typisierungsregeln von Damas und Milner nahe sind, die wir aus den anderen Seminarvoträgen kennen.

Literatur

[1] Cordelia V Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[2] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.