Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

Abgeleitete instance-Deklarationen für Summen-Typen

- Unsere `instance (Eq a) => Eq (Tree a)` Deklaration folgt der rekursiven Struktur des Datentyps `Tree`.

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
instance (Eq a) => Eq (Tree a) where
  (==) t1 t2 = case (t1,t2) of
     (Leaf x, Leaf y) -> (x==y)
     (Node tl1 k1 tr1, Node tl2 k2 tr2) ->
         (tl1==tl2) && (k1==k2) && (tr1==tr2)
     _ -> False
```

- Solche **instance-Deklarationen** für manche (vordefinierte) Typklassen können für Summentypen **automatisch abgeleitet** werden.

- Bsp.: Die abgeleitete `==`-Operation aus `Eq` für einen Summentyp benutzt die Identität der Konstruktoren und die rekursive Gleichheit der Komponenten.

Abgeleitete instance-Deklarationen für Summen-Typen

- Automatisch abgeleitete instance-Deklarationen werden mit folgender Syntax bei der Definition der Typen eingeführt.

```
data T α1 α2 ... αn = Konstruktor1 β11 ... β1n1
                    | Konstruktor2 β21 ... β2n2
                    ...
                    | Konstruktorm βm1 ... βmnm
        deriving (C1, ..., Cp)
```

- Bsp.:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
      deriving (Eq, Show)
```

Oberklassen

- Haskell unterstützt Klassenerweiterung: eine Typklasse C vererbt alle Operationen der Klassen C_1, C_2, ..., C_n.
  $\Longrightarrow$ ein Typ T ist eine Instanz der Typklasse C nur wenn er auch schon eine Instanz von C_1, ..., C_n ist.

- Syntax:

```
class (C1, ..., Cn) => C α where ...
```

- Bsp.:

```
class (Eq a) => Ord a where ...
```

- Die Klassenhierarchie, die dadurch entsteht muss azyklisch sein.

- In C_i darf lediglich α als Typvariable auftreten.

Vordefinierte Typklassen

- **Eq a**, wie oben
- **Ord a** mit `(<), (<=) :: a -> a -> Bool`
  - automatisch ableitbare instance-Dekl. für alle Summentypen laut Reihenfolge der Konstruktoren in `data`-Deklaration
- **Show a**, **Num a**, **Enum a**: siehe unten

Die vordefinierte Typklasse Show

- Für Typen deren Werte eine String-Darstellung haben:

```
class Show a where
  show :: a -> String
  showList :: [a] -> (String -> String)
  -- Default-Definition fuer showList v
```

- Instanzen müssen `show` definieren
- Wird benutzt u.a. von der Interpreterumgebung
- Automatisch ableitbare instance-Dekl. gemäß Konstruktornamen

Die vordefinierte Typklasse Num

- Alle numerischen Typen sind eine Unterklasse der Typklasse **Num**:

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

- Alle Instanzen müssen vergleichbar sein und eine String-Darstellung haben.
- Macht die Überladung numerischer Konstanten möglich: je nach Kontext kann `2` vom Typ `Integer` oder `Double` sein.
- Eine numerische Konstante k ist syntaktischer Zucker für `fromInteger` k.

Die vordefinierte Typklasse Enum

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
```

- Für Typen, deren Werte aufzählbar sind.
- Defaults für alle Operationen bis auf `toEnum` und `fromEnum`:
  - Bsp.: `succ = toEnum . (+1) . fromEnum`

Die vordefinierte Typklasse Enum

- Syntaktischer Zucker:
  - `[x..y]` ≡ `enumFromTo x y`

```
> [1 .. 8]
[1,2,3,4,5,6,7,8]
```

  - `[x..]` ≡ `enumFrom x`

```
> take 5 [3..]
[3,4,5,6,7]
```

  - `[x,y..z]` ≡ `enumFromThenTo x y z`

```
> [1,4 .. 20]
[1,4,7,10,13,16,19]
```

Die vordefinierte Typklasse Enum

- Automatisch ableitbare instance-Dekl. der Klasse `Enum` für Aufzählungstypen : gemäß Reihenfolge der Konstruktoren in data-Deklaration

- Bsp.:

```
data Wert =
  Zwei | Drei | Vier | Fuenf | Sechs | Sieben | Acht |
  Neun | Zehn | Bube | Dame | Koenig | As
 deriving (Eq,Ord,Enum,Show)

> [Sieben .. Dame]
[Sieben,Acht,Neun,Zehn,Bube,Dame]
```

List-Comprehensions

- List-Comprehensions sind syntaktischer Zucker für die Darstellung von Listen ähnlich wie Mengendefinitionen in der Mathematik:

$$A = \{f(x) \mid x \in B \wedge p(x)\}$$

- Bsp.:
  - Mathematische Notation:

    $Even = \{x \mid x \in \mathbb{N}, x \bmod 2 == 0\}$

  - Haskell-Notation:

```
even = [ x | x <- [1 ..], x `mod` 2 == 0]
```

```
> take 4 even
[2,4,6,8]
```

List-Comprehensions

- **Syntax**: `[ e | `q_1`, `q_2`, ..., `q_n`]` mit $n \geq 1$ und die Qualifier q_i einer der folgenden Formen sind:
  - **Generatoren**: *pattern*$_i$ `<-` *expr*$_i$ mit *expr*$_i$ vom Typ $[\alpha_i]$
  - **Filter**: Ausdrücke vom Typ `Bool`
- **Semantik** (operational):

```
[ e | v <- [], qs ]      → []
[ e | v <- (x:xs), qs ] →
    [ e | qs ][x/v] ++ [ e | v <- xs, qs ]
[ e | False, qs ]        → []
[ e | True, qs ]         → [ e | qs ]
[ e | ]                  → [ e ]
```

List-Comprehensions: Beispiele

```
> [(x,y) | x <- [1,2,3], y <- [4,5,6]]
[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]

> [(x,y) | x <- [1,2,3], y <- [4,5,6], x+y > 6]
[(1,6),(2,5),(2,6),(3,4),(3,5),(3,6)]

> [x | x <- [-100..100], x^2 + x -12 == 0]
[-4,3]

pythTriples n = [(x,y,z) | x <- [1.. n],
                           y <- [x..n],
                           z <- [y..n],
                           x^2 + y^2 == z^2]
> pythTriples 10
[(3,4,5),(6,8,10)]

quick l = case l of
  [] -> []
  h:r -> (quick [x | x <- r, x <= h]) ++ [h]
         ++ (quick [x | x <- r, x > h])
```

Unendliche Datenstrukturen

- Die verzögerte Auswertung erlaubt unendliche Datenstrukturen direkt darzustellen.

- Bsp.:

```
onlyOnes = 1 : onlyOnes

natsFrom n = n : natsFrom (n+1)
-- synt. Zucker [n..]

squares = [ x^2 | x <- [1..]]

addCrtNumber l = zip [1..] l
```

Part III

Logische Programmierung

Logik als Programmiersprache

- Logik wird als Programmiersprache benutzt.
- Der logische Ansatz zu Programmierung ist (sowie der funktionale) deklarativ.
- Programme können mit Hilfe zweier abstrakten, Maschinen-unabhängigen Konzepte verstanden werden:
  - **Wahrheit**
  - **Logische Deduktion**

Deklarativität der logischen Programmierung

- Das auszuführende **Programm** wird spezifiziert durch:
  - das Wissen über ein Problem und die Annahmen, die hinreichend für die Lösung sind $\equiv$ **logische Axiomen**;
  - eine zu beweisende Aussage (**Ziel**, **goal statement**) als das zu lösende Problem. ($\approx$ Eingabe)
- Die **Ausführung**:
  - ist der Versuch das goal statement zu beweisen unter den gegebenen Annahmen.

Hauptkonstrukte

Die Hauptkonstrukte der logischen Programmierung stammen aus der Logik:

- **Aussagen**: Fakten, Anfragen und Regeln
- **Terme** $\equiv$ die einzigen Datenstrukturen

Fakten

- **Fakten** sind Aussagen, die Beziehungen zwischen Objekten definieren.
- Bsp.:

```
father(abraham,isaac).
```

  ... besagt, dass zwischen `abraham` und `isaac` die Beziehung `father` besteht.

- Eine Beziehung kann man als ein **Prädikat** auffassen: das Prädikat `father` gilt für `abraham` und `isaac`.

Fakten

- Die *Plus*-Beziehung:

```
plus(0,0,0).  plus(0,1,1).  plus(0,2,2).  plus(0,3,3).
plus(1,0,1).  plus(1,1,2).  plus(1,2,3).  plus(1,3,4).
plus(2,0,2).  plus(2,1,3).  plus(2,2,4).  plus(2,3,5).
```

- Eine endliche Menge von Fakten ist das einfachste Programm.

Anfragen

- **Anfragen** (*queries*) erlauben es zu testen, ob eine Beziehung zwischen Objekten besteht, und somit Informationen aus einem Programm abzufragen.

- Bsp.:

```
father(abraham,isaac)?
```

  ... fragt, ob die Beziehung `father` zwischen `abraham` und `isaac` besteht.

Anfragen

- Um eine **Anfrage** für ein gegebenes Programm **zu beantworten**, muss man **bestimmen**, **ob** die **Anfrage** eine **logische Folge des Programms** (d.h. der spezifizierten Axiomen) laut der **Deduktionsregeln** ist.

- Die einfachste Deduktionsregel ist **Identität**:

  aus P **folgt** P

  D.h. eine Anfrage ist die logische Folge eines identischen Fakten.

- Bsp.: `father(abraham,isaac)?` ist wahr, angenommen der Fakt `father(abraham,isaac).` Teil des Programms ist.

Logische Variablen

- Statt nur `wahr` oder `falsch` als Antworten zu bekommen, möchte man manchmal auch Objekte mit einer bestimmten Eigenschaft herausfinden.

- Bsp.: *Wessen Vater ist Abraham?*
  - 1. Möglichkeit: Wiederholte Anfragen

    `father(abraham,lot)?`, `father(abraham,milcah)?`, ..., `father(abraham,isaac)?`, ...

    bis man die Antwort `wahr` erhält.

  - 2. Möglichkeit: Besser, benutze **Variablen** , um über unbekannte Objekte zu sprechen...

Terme

- Die Objekte eines Programms werden als **Terme** spezifiziert. Induktive Definition:
  - **Atome:** Konstanten (z.B. `abraham`, `lot`, `0`, `1`) sind Terme.
  - **Variablen:** Variablen (z.B `X`, `Y`) sind Terme.
  - **Zusamengesetzte Terme:** Wenn t_1, t_2, ..., t_n Terme sind, und f ein Name ist, dann ist $f(t_1, t_2, \ldots, t_n)$ ein Term.
    - f = **Funktor**
    - t_1 bis t_n = **Argumente**
    - n = **Stelligkeit** des Funktors
- Können als Bäume aufgefasst sein:
  - Blätter = Atome, Variablen
  - innere Knoten = Funktoren
- Syntaktische Konvention: Nur Variablennamen werden großgeschrieben.

Terme

- Beispiele:
  - `name(john,smith)`
  - `name(X,Y)`
  - `person(name(john,smith),age(23))`
  - `person(name(X,smith),age(23))`
  - `person(Y,age(23))`
  - `node(node(leaf(a),leaf(b)),leaf(c))`
  - `s(0)`
- Terme sind die einzige Datenstruktur in logischen Programmen.

Substitutionen und Instanzen

- Eine **Substitution** ist eine endliche Menge von Paaren der Form $X_i = t_i$, mit X_i eine Variable, t_i ein Term und:
  - $X_i \neq X_j \ \forall i \neq j$
  - X_i tritt in keinem t_j auf $\forall i, j$

  Bsp.: `{X=isaac}`

- Die **Anwendung einer Substitution** θ auf einen Term A, $A\ \theta$, ist der Term, den man erhält, indem man für jedes Paar $X = t$ in θ jedes Auftreten von X durch t ersetzt .

- A ist eine **Instanz** von B, wenn eine Substitution θ existiert, so dass $A = B\ \theta$.
  - Bsp.: `father(abraham,isaac)` ist eine Instanz von `father(abraham,X)` unter der Substitution `{X=isaac}`.

Variablen in Anfragen

- **Existentielle Anfragen** = Anfragen mit Variablen
  - Bsp.: `father(abraham,X)?`
  - Haben eine **existentielle** Interpretation:
    Existiert *eine Substitution, für welche die Anfrage eine logische Folge des Programms ist?*
- **Verallgemeinerung** als Deduktionsregel:

  Eine existentielle Anfrage $P?$ ist eine Folge einer Instanz $P\ \theta$ für jedes θ.
  - Bsp.: `father(abraham,X)?` ist eine Folge von `father(abraham,isaac).`

Variablen in Anfragen

- Bsp.: `father(abraham,X)?`
- Der Beweis einer Anfrage ist **konstruktiv**, d.h. falls die Anfrage mit *ja* beantwortet werden kann, wird eine Substitution ausgegeben, für die die Aussage aus dem Programm deduzierbar ist.
- *Antwort*: *yes,* $\{X \mapsto isaac\}$

Variablen in Fakten

- Variablen in Fakten sind **universal quantifiziert**. Ein Fakt $p(t_1, t_2, \ldots, t_n)$ besagt, dass **für jedes** $X_1, \ldots, X_k$, mit X_i eine Variable die im Fakt auftritt, $p(t_1, \ldots, t_n)$ wahr ist.
  - Bsp.: `likes(X,pomegranates)` besagt, dass für alle `X`, `X` Granatäpfel mag.

- **Instantiierung** als Deduktionsregel:

  Aus einer universell quantifizierten Aussage P folgt eine Instanz $P\ \theta$ für jede Substitution θ.

  - Bsp.: `likes(lot,pomegranates)` folgt aus dem Fakt `likes(X,pomegrantes).`

Konjunktive Anfragen

- Eine Aussage $p(t_1, t_2, \ldots, t_n)$ heißt auch **Ziel** (engl. *goal*).
- Eine Anfrage $p(t_1, t_2, \ldots, t_n)?$, die aus nur einem Ziel besteht heißt **einfach**.
- Eine Anfrage kann auch eine Konjunktion aus mehreren Zielen sein $\Longrightarrow$ **konjunktive Anfragen**.
- Eine konjunktive Anfrage ist die Folge eines Programms, wenn jedes Ziel eine Folge des Programms ist.
- Bsp. `father(abraham,isaac),male(lot)?`