Funktionell programmering

DD1361

Polymorfism

- Polymorf funktion: Kan tillämpas på flera typer.
- Polymorf typ: ”Kan anta flera former” (Hutton).
- Vilken typ har `length`? Kika!

```
Prelude>:t length
length :: [a] -> Int
```

- `a` är en typvariabel
- Terminologi: parametrisk polymorfism

Fler polymorfa exempel

```
Prelude> :type head
head :: [a] -> a

Prelude> :t tail
tail :: [a] -> [a]

Prelude> :t fst
fst :: (a,b) -> a
```

Typklasser

- Hur ställer vi krav på typer?
- Om `sum :: [a] -> a`, hur garanterar vi att addition är definierat på a?
- Om `sort :: [a] -> [a]`, hur vet vi att <= finns på a?

```
Prelude> :t sum
sum :: Num a => [a] -> a
Prelude> :t sort
sort :: Ord a => [a] -> [a]
```

Några typklasser

- Num Alla numeriska typer
- Ord Alla ordnade typer (<=)
- Eq Alla jämförbara typer (==)
- Show Typer med skrivbara element
- Read Typer med läsbara element
- Integral Heltalstyper: `Int, Integer`
- Fractional `Float` tex, de som stödjer division.

Överlagring (overloading)

- Samma funktion definierad på olika argument
- T.ex.: Addition (+) på `Int` är inte samma som addition på `Float`.
- ”ad hoc” polymorfism
- C++:

```
int Klass::kamp(int x, int y) {
return x * y; }
int Klass::kamp(String x, String y) {
return x.length() * y.length(); }
```

Överlagring i Haskell

- Explicit överlagring
- Skapa en klass som beskriver gemensamma egenskaper
- Visa hur överlagringen hanteras för olika typer

Överlagring i Haskell

- Från Prelude.hs:

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

- Också från Prelude.hs:

```
instance Eq Integer where
x == y = integerEq x y
```

- Tala om att din klass tillhör Eq:

```
instance Eq MinKlass where
x == y = minEq x y
```

Arv hos typklasser

- Definiera ordnade typer:

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

- ”Alla `Ord`-typer måste vara `Eq`-typer också.”

Tillämpning

```
class Incrementable a where
      inc :: a -> a
instance Incrementable Char where
      inc c = chr ((ord c) + 1)
instance Incrementable a =>
   Incrementable [a] where
        inc l = map inc l
Prelude> inc ’a’
’b’
Prelude> inc "HAL"
"IBM"
```

Klassfunktioner

En klass kan definiera gemensamma funktioner:

```
class Incrementable a where
      inc :: a -> a
      inc2 :: a -> a
      inc2 x = inc (inc x)
instance Incrementable Char where
      inc c = chr ((ord c) + 1)
Prelude> inc ’a’
’b’
Prelude> inc2 ’a’
’c’
```

Omdefinition

```
class Incrementable a where
      inc :: a -> a
      inc2 :: a -> a
      inc2 x = inc (inc x)
instance Incrementable Integer where
      inc x = x + 1
instance Incrementable Char where
      inc c = chr ((ord c) + 1)
      inc2 c = chr ((ord c) - 32)
```

Omdefinition

```
Prelude> inc2 1
3
Prelude> inc ’a’
’b’
Prelude> inc2 ’a’
’A’
```

Jämför med objektorientering

Likheter

- Klasser och arv
- Klasser samlar funktioner
- Standardmetoder

Skillnad

- Inga objekt, inga attribut
- Två sorts polymorfism: Parametrisk och ad hoc
- Alla metoder är statiskt bestämda, ingen klassinformation i data
- Åtkomstkontroll (public, private) indirekt via modulsystem

Typkonvertering

I många språk:

```
Pseudo> length(lista) / 10
⇒5.5
```

I Haskell:

```
Prelude> length(lista) / 10
<interactive>:1:0:
No instance for (Fractional Int)
arising from a use of ‘/’ at
Possible fix: add an instance
declaration for (Fractional Int)
```

Explicit typkonvertering

```
Prelude> fromIntegral(length(lista))/ 10
5.5
Prelude> :t fromIntegral
fromIntegral :: (Integral a, Num b)
=> a -> b
```

Typkonverteringar i Prelude

- fromInteger
- fromRational
- toInteger
- toRational
- fromIntegral
- fromRealFrac
- fromIntegral
- fromRealFrac

Definiera typer

God programmering kräver

- god organisering
- god abstraktion

Abstrahera och organisera dina data!

I Haskell

- typer
- datastrukturer

Definera egna typnamn

```
type Distance = Float
```

gör att `Float` och `Distance` kan användas omväxlande.

Liknande med

```
type Coord = (Float, Float)
```

Definera egna typnamn

Även med polymorfa typer:

```
type AssocList a b = [(a, b)]
```

Användning:

```
getElem :: AssocList String Int ->
String -> Int
```

Fördelar:

Praktiskt! Tydliggörande!

Definiera egna datastrukturer

Vi vill kunna aggregera information.

Pascal, C, etc: record eller struct

Java, C++, etc: Klasser

Haskell: `data`

Exempel:

```
data Bool = True | False
data Address = None | Addr String
```

Konstruktorer: `True, False, None, Addr`

Exempel: Datatyp för färger

```
data Color = RGB Int Int Int
black = RGB 0 0 0
white = RGB 256 256 256
Main*> :t black
black :: Color
Main*> black
```

Ger felmeddelande! Detta händer även när man jämför de två objekten, `black == white`

Exempel: Datatyp för färger

```
data Color = RGB Int Int Int
    deriving (Show, Eq) -- mer?

black = RGB 0 0 0
white = RGB 256 256 256
```

Med hjälp av deriving kommer vi åt klasserna Show och Eq

Exempel: Datatyp för färger

```
Main*> black
RGB 0 0 0
Main*> white
RGB 256 256 256
Main*> black == white
False
```

Polymorfa datatyper

Välj din egen färgrepresentation:

```
data Color f = RGB f f f
                 deriving (Show, Eq)

black = RGB 0 0 0
white = RGB 1.0 1.0 1.0
```

Polymorfa datatyper

...fast begränsa till nummer!

```
data Num f=>Color f = RGB f f f
                    deriving (Show, Eq)

black = RGB 0 0 0
white = RGB 1.0 1.0 1.0
```

Datatyper och mönstermatchning

```
data Color = RGB Float Float Float
             deriving (Show, Eq)
red (RGB r g b) = r
green (RGB r g b) = g
blue (RGB r g b) = b
brightness :: Color -> Float
brightness (RGB r g b) =
           sqrt((r^2+g^2+b^2) / 3)
```

Datatyper med åtkomstfunktioner

Alternativ definition av Color:

```
data Color =
     RGB {red, green, blue :: Float}
                    deriving (Show, Eq)
brightness :: Color -> Float
brightness c =
    sqrt(((red c)^2+(green c)^2+
          (blue c)^2)/3)
```

Algebraiska datatyper

- En typ ”komponerad” av andra typer, var

och en med en konstruktor, kallas algebraisk.

- Obs: En produkttyp har endast en konstruktor.

Ex: `data Color = RGB Int Int Int`

- Obs: En enumererad typ listar konstruktorer utan argument.

Ex: `data Bool = True | False`

Minns färgmodulen: Color

```
data Color = RGB Int Int Int
```

Hur göra om vi vill ändra representationen?

```
data Color = HSV Int Int Int
```

Förstör annan kod!

```
brightness :: Color -> Float
brightness (RGB r g b) =
                    sqrt((r^2+g^2+b^2) / 3)
```

Exempel: En ADT för färger

```
data Color = RGB Int Int Int

makeColorRGB r g b = RGB r g b
makeColorHSV h s v = ...
```

Typer i olika språk

Statisk typning: Typen begränsar variabeln:

```
int i := 17
```

Används i Java, Haskell, etc.

Dynamisk typning: Beskriver värdet. Tänk:

```
i := 17
```

Används i Lisp, MatLab, Perl, etc

Typkonvertering

Implicit konvertering:

```
float x = 3.5;
int i = x;
```

Explicit konvertering:

```
float x = 3.5;
int i = (int) x;
```

Konvertering och omtolkning

- Vad får i för värde?

```
char *s = "Hej!";
int i = (int) s;
```

- Vad händer här?

```
long int stort =0x7FFFFFFFFFFFFFFF;
int litet = (int) stort;
printf("stort = %ld, litet = %d\n",
        stort, litet);
```

Utdata:

```
stort = 9223372036854775807, litet = -1
```

Stark och svag typning

- Oklara begrepp. Avser hur starkt språket begränsar tolkningen av värden.
- **Sebesta:** ”strongly typed if all type errors are always detected.”
- ”Ada, Java, and C# are nearly strongly typed”

Stark och svag typning: exempel

- Tcl: Alla variabler kan tolkas som strängar.
- Perl: Listvariabel kan tolkas som listlängd, beroende på sammanhang. En lista kan tolkas som en associativ array (hashtabell).
- C: Automatisk typkonvertering (casting) mellan flera inbyggda typer. Union kan ej typkontrolleras.
- C++: Skriv dina egna automatiska konverteringsregler.
- Haskell: Ingen automatisk typkonvertering.

Säkra typsystem

- Def: Ett typsystem är säkert om det garanterar att inga uttryck kan anta värden som inte är definierade av typen.
- Osäkert språk: C
- Säkra språk: Haskell, Ada(?)