Abstrakte Datentypen in Haskell

Wolfgang Ginolas, Stefan Schmidt

Fachhochschule Wedel

11. Dezember 2007

Inhalt

Einführung Abstrakte Datentypen (ADTs)

Abstrakter Datentyp (abstract data type / ADT):

- Menge von Objekten (Werte)
- Operatoren auf diesen Werten (Schnittstelle)
- Genaue Beschreibung der Semantik der Operatoren
- Keine Aussage über die konkrete Implementierung der Operatoren
- Meist parametrisierbar (parameterized data type)

Konkreter Datentyp (concrete data type):

- Festlegung der Implementierung

Formale Beschreibung von ADTs

Um Operatoren benutzen zu können, muss folgendes bekannt sein:

- Name des Operators (Syntax)
- Bedeutung des Operators und seiner Beziehung zu anderen (Semantik)

Beschreibung der Syntax:

- Schnittstelle des Datentyps

Beschreibung der Semantik:

- Informelle Beschreibung
- Algebraische Strukturen
- Axiome

Eigenschaften von ADTs

Universalität (implementation independence)

- In beliebigen Programmen einsetzbar

Präzise Beschreibung (precise specification)

- Schnittstelle eindeutig und vollständig

Einfachheit (simplicity)

- Kein Wissen über interne Implementierung notwendig

Kapselung (encapsulation)

- Interne Implementierung vor dem Benutzer verborgen

Geschütztheit (integrity)

- Keinen Zugriff auf interne Implementierung

Modularität (modularity)

- Möglichst kleine ADTs, dadurch einfacher Austausch

Queue - Interface

```
empty     :: Queue a
isEmpty   :: Queue a -> Bool
join      :: a -> Queue a -> Queue a
front     :: Queue a -> a
back      :: Queue a -> Queue a
```

Queue - Axiome

```
isEmpty empty                  = True
isEmpty (join x xq)            = False

front (join x empty)           = x
front (join x (join y xq)      = front (join y xq)

back (join x empty)            = empty
back (join x (join y xq))      = join x (back (join y xq))

isEmpty (join x undefined) = isEmpty undefined = undefined
```

Modulkonzept von Haskell

Modul:

- Unterteilung eines Programms in Funktionseinheiten
- Voraussetzung, um in Haskell ADTs implementieren zu können
- Bilden eigenen Namensraum (namespace)

```
module Queue (Queue, empty, isEmpty, join, front, back) where

import Test.HUnit

newtype Queue a = MkQ([a], [a])
empty :: Queue a
empty = MkQ([],[])

isEmpty :: Queue a -> Bool
isEmpty (MkQ (xs, ys)) = null xs
```

Modulkonzept von Haskell

- Nur ein Modul pro Datei
- Modulname beginnt mit einem Großbuchstaben
- Hierarchie:
  - Haskell98: flach
  - Heute: Bildung von “Untermodulen” durch Punkt-Notation
- Keine zyklische Abhängigkeit zwischen Modulen

Import von Modulen

- Default: gesamte Schnittstelle importieren
- Einschränkung durch explizite Angabe

Export aus Modulen

- Default: gesamte Schnittstelle exportieren
- Einschränkung durch explizite Angabe

Entstehung von Mehrdeutigkeiten

```
module MyMod1 where              -- alles exportieren

kick :: String -> String
kick str = filter (/='a') str
```

```
module MyMod2 (kick) where  -- nur "kick" exportieren

kick :: String -> String
kick str = map (\c -> if c == 'a' then '_' else c) str
```

```
import MyMod1 (kick)          -- nur "kick" importieren
import MyMod2                 -- alles Importieren

myFunc :: String -> String
myFunc = reverse . kick       -- Kick aus MyMod1 oder MyMod2 ?
```

Auflösen von Mehrdeutigkeiten

```
-- Auflösen des Konfliktes durch explizite Angabe
myFunc2 :: String -> String
myFunc2 = reverse . MyMod2.kick
```

```
-- Erzwingen der Punkt-Notation
import qualified MyMod1
import qualified MyMod2
```

```
-- Verstecken von "kick" aus MyMod1
import MyMod1 hiding (kick)
import MyMod2
```

Re-Export von Modulen

Re-Import von Modulen:

- Zusammenfassen von mehreren Modulen in einer Schnittstelle
- Sinnvoll bei der Erstellung von Bibliotheken

```
module MyLibrary ( module MyMod1, module MyMod3 ) where

import MyMod1
import MyMod3
```

Set - Interface

```
empty    :: Set a
isEmpty  :: Set a -> Bool
member   :: Set a -> a -> Bool
insert   :: a -> Set a -> Set a
delete   :: a -> Set a -> Set a

union    :: Set a -> Set a -> Set a
meet     :: Set a -> Set a -> Set a
minus    :: Set a -> Set a -> Set a
```

Set - Axiome I

```
insert x (insert x xs) = insert x xs
insert x (insert y xs) = insert y (insert x xs)

isEmpty empty          = True
isEmtpy (insert x xs)  = False

member empty y         = False
member (insert x xs) y = (x == y) || member xs y

delete x empty         = empty
delete x (insert y xs) =
  if x == y
  then delete x xs
  else insert y (delete x xs)
```

Set - Axiome II

```
union xs empty          = xs
union xs (insert y ys) = insert y (union xs ys)

meet xs empty           = emtpy
meet xs (insert y ys)   =
  if member xs y
  then insert y (meet xs ys)
  else meet xs ys

minus xs empty          = xs
minus xs (insert y ys) = minus(delete y xs) ys
```

Bag - Interface

```
empty    :: Bag a
insert   :: a -> Bag a -> Bag a
mkBag    :: [a] -> Bag a
isEmpty  :: Bag a -> Bool
union    :: Bag a -> Bag a -> Bag a
minBag   :: Bag a -> a
delMin   :: Bag a -> Bag a
```

Bag - Axiome I

```
isEmpty (mkBag xs)                  = null xs

union (mkBag xs) (mkBag ys)         = mkBag(xs ++ ys)

minBag (mkBag xs)                   = minlist xs

delMin (mkBag xs)                   = mkBag (deleteMin xs)
```

Bag - Axiome II

```
insert x (insert y xb)           = insert y (insert x xb)

mkBag                            = foldr insert empty

union xb empty                   = xb
union xb (insert y yb)           = insert y (union xb yb)

minBag (insert x empty)          = x
minBag (insert x (insert y xb)) = x `min` minBag (insert y xb)

delMin (insert x empty)          = empty
insert (minBag xb) (delMin xb)   = xb
```

Flexible Array - Interface

```
empty     :: Flex a
isEmpty   :: Flex a -> Bool
access    :: Flex a -> Int -> a
update    :: Flex a -> Int -> a -> Flex a
hiext     :: a -> Flex a -> Flex a
hirem     :: Flex a -> Flex a
loext     :: a -> Flex a -> Flex a
lorem     :: Flex a -> Flex a

size      :: Flex a -> Int
```

Flexible Array - Axiome

```
hiext x . loext y                  = loext y . hiext x

hirem empty                        = error
hirem (hiext x xf)                 = xf
hirem (loext x empty)              = empty
hirem (loext x (hiext y xf))       = loext x xf
hirem (loext x (hiext y xf))       = loext x (hirem (loext y xf))

access empty k                     = error
access (loext x xf) 0              = x
access (loext x xf) (k + 1)        = access xf k

access (hiext x xf) k
  | k < n = access xf k
  | k == n = n
  | k > n = error
  where n = size xf
```

Vordefinierte ADTs

Im Data-“Package”

- Data.List
- Data.Tuple
- Data.Map
- Data.HashTable
- Data.Array
- Data.Set
- Data.Tree
- ...

Quellen I

R. Bird
Introduction to Functional Programming using Haskell
Prentice Hall, 1998

Haskell Project Homepage
Haskell
`http://www.haskell.org`

Wikibooks.org
Haskell
`http://en.wikibooks.org/wiki/Haskell`