GADTs

Generalisierte Algebraische Datentypen

Sven Urbanski (svenurbanski@gmail.com)

Sommersemester 2011

1. Einführung
   - Motivation
   - Von ADTs zu GADTs
2. Erweiterungen durch GADTs
   - Phantomtypen
   - Typeinschränkungen
   - Erweiterung der Rückgabetypen
3. Beispiele für GADTs
   - Sichere Listen
   - Gemischte Listen
   - Abstrakte Syntaxbäume
   - Vektoren
4. Schlussbetrachtungen
   - Fazit
   - Verwandtes: Associated Types
   - Verwandtes: Dependent Types

Motivation

Generalisierte Algebraische Datentypen
(GADT, [gʌt]) bieten:

- ⇒ Formulierung von Invarianten im Typsystem
- ⇒ Zusätzliche Typchecks
- ⇒ weniger Laufzeitfehler

ADT

⇒ **Algebraische Datentypen**
([a:de:te:] auch **Summen-Datentypen**):

Beispiel:

```
1  data X = X1 A | X2 B | ...
```

⇒ X ist der ADT
X1, X2, ... sind Konstruktorfunktionen
X ≈ die Menge aller Werte aus A, B, ...
$X \approx \{a1, a2, a3, \ldots, b1, b2, b3, \ldots\}$
$|X| = |A| + |B| + \ldots$

Anwendungsfälle

$\Rightarrow$ beliebig viele

$\Rightarrow$ überall, wo im Datentyp verschiedene Fälle unterschieden werden

Beispiel:

```
1  data Expr = IExpr Int
2            | BExpr Bool
3            | AndExpr Expr Expr
```

Phantomtypen

Normale Typparameter:

```
1  data List a = Nil
2              | Cons a (List a)
```

Typparameter, die scheinbar nicht genutzt werden:

```
1  data PList a b = PNil
2                 | PCons a (PList a b)
```

Phantomtypen

Zur Erinnerung:

```
1 data PList a b = PNil
2                | PCons a (PList a b)
```

Vorteil: Listen mit unterschiedlicher Semantik können nicht mehr gemischt werden

Beispiel:

```
1 data Euro = ...
2 data USD = ...
3 euroToUsd :: PList a Euro -> PList a USD
```

Einschränkungen der Typparameter

In Java:

```
1 interface NumList<A extends Number>
2 class NCons<A> implements NumList<A>
3 class NNil ...
```

In Haskell:

```
1 data NumList a = NNil
2                | Num a => NCons a (NumList a)
```

Syntax

Alte Syntax:

```
1  data List a = Nil
2              | Cons a (List a)
```

Neue Syntax:

```
1  data List a where
2    Nil :: List a
3    Cons :: a -> (List a) -> List a
```

Nil ist eine Funktion ohne Parameter.
Cons ist eine Funktion mit zwei Parametern.
Gemeinsamkeiten?

Erweiterung durch GADTs

Willkürliche Beispiele:

```
1  Cons1 :: a -> (List a) -> List a
2  Cons2 :: a -> (List a) -> List (a,a)
3  Cons3 :: a -> (List a) -> List (List a)
4  Cons4 :: a -> (List a) -> a -- err
```

⇒ Welchen Vorteil bringt das?

⇒ Mehr Typchecks

⇒ Aber wie sinnvoll einsetzen?

Sichere Listen

⇒ Was bedeutet "sicher"?

⇒ **Typ**-Unterscheidung zur Compilezeit:
Leere / Nicht leere Listen

⇒ Sinnvoll für partiell definierte Funktionen wie `head` und `tail`

Beispiel:

```
1 data Safe a  = Safe a
2 data NotSafe = NotSafe
3 data SafeList a b where
4   Nil  :: SafeList a NotSafe
5   Cons :: a -> SafeList a b -> SafeList a (Safe b)
```

Typ von sicheren Listen

Zur Erinnerung:

```
1 data SafeList a b where
2   Nil  :: SafeList a NotSafe
3   Cons :: a -> SafeList a b -> SafeList a (Safe b)
```

Bedeutung der Typparameter:

```
1 SafeList a NotSafe                  -- leere Liste
2 SafeList a (Safe NotSafe)           -- Laenge == 1
3 SafeList a (Safe b)                 -- Laenge >= 1
4 SafeList a (Safe (Safe NotSafe))    -- Laenge == 2
5 SafeList a (Safe (Safe b))          -- Laenge >= 2
6 SafeList a b                        -- Laenge beliebig
7 ...                                 -- Laenge n ???
8
```

Funktionen mit sicheren Listen: safeHead

Zur Erinnerung:

```
1 data SafeList a b where
2   Nil  :: SafeList a NotSafe
3   Cons :: a -> SafeList a b -> SafeList a (Safe b)
```

Sicher:

```
1 safeHead :: SafeList a (Safe b) -> a
2 safeHead (Cons x _) =  x
3 -- safeHead Nil = ???
```

Funktionen mit sicheren Listen: unsafeHead

Zur Erinnerung:

```
1 data SafeList a b where
2   Nil  :: SafeList a NotSafe
3   Cons :: a -> SafeList a b -> SafeList a (Safe b)
```

Unsicher:

```
1 unsafeHead :: SafeList a b -> a
2 unsafeHead (Cons x _) =  x
3 unsafeHead Nil = error "error in head!"
```

Funktionen mit sicheren Listen: tail

Zur Erinnerung:

```
1 data SafeList a b where
2   Nil  :: SafeList a NotSafe
3   Cons :: a -> SafeList a b -> SafeList a (Safe b)
```

Beispiel:

```
1 safeTail :: SafeList a (Safe b) -> SafeList a b
2 safeTail (Cons _ t) = t
```

Funktionen mit sicheren Listen: access

Zur Erinnerung:

```
1 data SafeList a b where
2   Nil  :: SafeList a NotSafe
3   Cons :: a -> SafeList a b -> SafeList a (Safe b)
```

Sicherer Zugriff?

```
1 accessN :: Int
2            -> SafeList a (Safe (Safe (Safe ...)))
3            -> a
```

Funktionen mit sicheren Listen: map/fold

⇒ Was ist mit total definierten Funktionen wie map oder fold?

⇒ Kein Problem!

map:

```
1  safeMap :: SafeList a c -> (a -> b) -> SafeList b c
2  safeMap (Cons e t) f = Cons (f e) (safeMap t f)
3  safeMap (Nil) _ = Nil
```

fold:

```
1  fold :: (a -> b -> a) -> a -> SafeList b c -> a
2  fold f start (Nil) = start
3  fold f start (Cons e t) = fold f (f start e) t
```

Funktionen mit sicheren Listen: zip

Zur Erinnerung:

```
1 data SafeList a b where
2   Nil  :: SafeList a NotSafe
3   Cons :: a -> SafeList a b -> SafeList a (Safe b)
```

zip:

```
1 safeZip :: SafeList a len -> SafeList b len
2         -> SafeList (a,b) len
3 safeZip Nil Nil = Nil
4 safeZip (Cons e1 l1) (Cons e2 l2) =
5     Cons (e1,e2) (safeZip l1 l2)
```

Funktionen mit sicheren Listen: Konvertierung 1

Zur Erinnerung:

```
1 data SafeList a b where
2   Nil  :: SafeList a NotSafe
3   Cons :: a -> SafeList a b -> SafeList a (Safe b)
```

Unsichere Liste → Sichere Liste

```
1 makeSafe1 :: SafeList a b
2              -> Maybe (SafeList a (Safe b))
3 makeSafe1 Nil = Nothing
4 makeSafe1 (Cons e l) = Just (Cons e l)  --- ???
```

Funktionen mit sicheren Listen: Konvertierung 2

Zur Erinnerung:

```
1 data SafeList a b where
2   Nil  :: SafeList a NotSafe
3   Cons :: a -> SafeList a b -> SafeList a (Safe b)
```

Unsichere Liste → Sichere Liste

```
1 makeSafe2 :: SafeList a b
2              -> a
3              -> SafeList a (Safe b)
4 makeSafe2 l e = Cons e l
```

Möglich, aber identisch zu Cons

Funktionen mit sicheren Listen: Konvertierung 3

Zur Erinnerung:

```
1 data SafeList a b where
2   Nil  :: SafeList a NotSafe
3   Cons :: a -> SafeList a b -> SafeList a (Safe b)
```

Sichere Liste → Unsichere Liste

```
1 makeUnsafe :: SafeList a (Safe b)
2               -> SafeList a b
3 makeUnsafe (Cons e l) = Cons e l -- ???
```

Sinnvoll?

Gemischte Listen

Idee: Verschiedene Typen in einer Liste

Beispiel:

```
1  data MixList a b where
2    MNil  :: MixList a ()
3    MCons :: (Show c, Show b) =>
4          a -> MixList b c -> MixList a (MixList b c)
```

$\Rightarrow$ a ist der Typ des ersten Elements

$\Rightarrow$ b ist der Typ der Restliste

Gemischte Listen

Zur Erinnerung:

```
1  data MixList a b where
2    MNil  :: MixList a ()
3    MCons :: (Show c, Show b) =>
4          a -> MixList b c -> MixList a (MixList b c)
```

Beispiel:

```
1  > MCons 42 (MCons "hallo" MNil)
2  42 : "hallo" : []
3  > :t MCons 42 (MCons "hallo" MNil)
4  MixList t (MixList [Char] (MixList b ()))
```

Unsichere Ausdrucksbäume (ADT)

ADT:

```
1  data Expr = IntLiteral Int
2            | BoolLiteral Bool
3            | AndExpr Expr Expr
4  inv :: Expr -> Bool
5  inv (BoolLiteral _) = True
6  inv (IntLiteral _)  = True
7  inv (AndExpr l r)   = isInt l && isInt r
8                        `or` isBool l && isBool r
9  -- isBool, isInt ...
```

⇒ Umständliche Definition der Invariante (bei vielen Konstruktorfunktionen)

⇒ Kann mit ADT nur zur Laufzeit überprüft werden

⇒ Ziel: Invariante durch Compiler überprüfen lassen

Sichere Ausdrucksbäume (GADT)

GADT:

```
1  data Expr a where
2    I      :: Int -> Expr Int
3    B      :: Bool -> Expr Bool
4    And    :: Expr Bool -> Expr Bool -> Expr Bool
```

- ⇒ Invariante?
- ⇒ Langweilig: Ausdrucksbäume bestehen nur aus Ints oder nur aus Bools
- ⇒ Erweiterung: Vergleich und Addition

Sichere Ausdrucksbäume (GADT)

Erweiterung:

```
1  data Expr a where
2    I     :: Int -> Expr Int
3    B     :: Bool -> Expr Bool
4    And   :: Expr Bool -> Expr Bool -> Expr Bool
5    Equal :: (Show a, Eq a) =>
6             Expr a -> Expr a -> Expr Bool
7    Add   :: Expr Int -> Expr Int -> Expr Int
```

$\Rightarrow$ Mischung von Ints und Bools **und** Invariante erfüllt

Sichere Ausdrucksbäume: eval

Zur Erinnerung:

```
1 data Expr a where
2   I     :: Int -> Expr Int
3   B     :: Bool -> Expr Bool
4   And   :: Expr Bool -> Expr Bool -> Expr Bool
5   Equal :: Eq a => Expr a -> Expr a -> Expr Bool
6   Add   :: Expr Int -> Expr Int -> Expr Int
```

Auswertung:

```
1 eval :: Expr a -> a
2 eval (I i)        = i
3 eval (B b)        = b
4 eval (And l r)    = eval l && eval r
5 eval (Equal l r) = eval l == eval r
6 eval (Add l r)    = eval l + eval r
```

Sichere Ausdrucksbäume: optimize

Zur Erinnerung:

```
1  data Expr a where
2    I     :: Int -> Expr Int
3    B     :: Bool -> Expr Bool
4    And   :: Expr Bool -> Expr Bool -> Expr Bool
5    Equal :: Eq a => Expr a -> Expr a -> Expr Bool
6    Add   :: Expr Int -> Expr Int -> Expr Int
```

optimize:

```
1  optimize :: Expr a -> Expr a
2  optimize (And (B True) r) = optimize r
3  optimize (And l r) = And (optimize l) (optimize r)
4  optimize x = x
```

Vektoren

Beispiel:

```
1  data Zero
2  data Next a
3  data Vec e n where
4    NullVec :: Vec e Zero
5    NextVec :: e -> Vec e n -> Vec e (Next n)
```

Konvertierung:

```
1  v2l :: Vec e n -> [e]
2  v2l NullVec = []
3  v2l (NextVec e l) = e : v2l l
4
5  l2v :: [e] -> Vec e n -- ???
6  l2v []     = NullVec
7  l2v (e:l) = NextVec e (l2v l)
```

Vektorfunktionen: Skalarprodukt

Zur Erinnerung:

```
1  data Zero
2  data Next a
3  data Vec e n where
4    NullVec :: Vec e Zero
5    NextVec :: e -> Vec e n -> Vec e (Next n)
```

Skalarprodukt:

```
1  dot :: Vec Int n -> Vec Int n -> Int
2  dot NullVec NullVec = 0
3  dot (NextVec e1 l1) (NextVec e2 l2) = e1 * e2 + dot l1 l2
```

Vektorfunktionen: Konkatenation

Zur Erinnerung:

```
1  data Zero
2  data Next a
3  data Vec e n where
4    NullVec :: Vec e Zero
5    NextVec :: e -> Vec e n -> Vec e (Next n)
```

Konkatenation:

```
1  concatV :: Vec e n1 -> Vec e n2 -> Vec e n3 -- ???
2  concatV NullVec x          = x
3  concatV (NextVec e1 l1) x = NextVec e1 (concatV l1 x)
```

Vektoren – zweiter Versuch:

Beispiel:

```
1  data Wec e n where
2    Wec :: [e] -> Wec e n
```

Konvertierung:

```
1  w2l :: Wec e n -> [e]
2  w2l (Wec l) = l
3
4  l2w :: [e] -> Wec e n
5  l2w l = Wec l
```

Vektoren – zweiter Versuch: Invariante

Zur Erinnerung:

```
1  data Wec e n where
2    Wec :: [e] -> Wec e n
```

Eigene Konstruktorfunktion:

```
1  w3 :: [e] -> Wec e (Next (Next (Next Zero)))
2  w3 e = assert (length e == 3) (Wec e)
```

Vektoren – dritter Versuch:

Beispiel:

```
1  data WecN e n where
2    Wec1 :: e -> WecN e (Next Zero)
3    Wec2 :: e -> e -> WecN e (Next (Next Zero))
4    Wec3 :: e -> e -> e -> WecN e (Next (Next (Next Zero)))
5    -- ...
```

$\approx$ Praktikabel?

Fazit

- $+$ Mehr Typsicherheit
- $+$ Weniger Laufzeitfehler
- $+$ Geschicktere Invarianten
- $+$ Partielle Funktion $\Rightarrow$ totale Funktion
- $\approx$ . . . aber nicht alle
- $\approx$ Es gibt sinnvolle Funktionen, die nicht definiert werden können
- $+$ Es gibt weniger sinnlose Funktionen, die definiert werden können

Associated Types (1)

⇒ Funktionen über Typen

⇒ Erweiterung von Typklassen

Herkömmliche Num-Typklasse:

```
1  class Num a where
2    (+++) :: a -> a -> a
3
4
5  instance Num Int where
6    (+++) x y = x + y
7
```

Neue Num-Typklasse:

```
1  class MyNum a b where
2    type Sum a b :: *
3    (+++) :: a -> b -> Sum a b
4
5  instance MyNum Int Int where
6    type Sum Int Int = Int
7    (+++) x y = x + y
```

⇒ Sum ist eine Funktion mit 2 Typen als Parameter und einem Typ als Rückgabewert

Associated Types (2)

Zur Erinnerung:

```
1  class MyNum a b where
2    type Sum a b :: *
3    (+++) :: a -> b -> Sum a b
```

MyNum Float Int:

```
1  instance Integral a => MyNum Float a where
2      type Sum Float a = Float
3      (+++) x y = x + intToFloat y
```

$\Rightarrow$ Floats und Ints addieren!

$\approx$ Ints und Floats?

Associated Types (3)

Zur Erinnerung:

```
1  class MyNum a b where
2    type Sum a b :: *
3    (+++) :: a -> b -> Sum a b
```

MyNum Int Float:

```
1  instance Integral a => MyNum a Float where
2      type Sum a Float = Float
3      (+++) x y = y +++ x
```

$\Rightarrow$ nun ist $+++$ kommutativ!

Dependent Types (1)

$\Rightarrow$ Typen, die von Werten abhängen

$\Rightarrow$ Agda

Dependent Types (2)

Willkürliches Beispiel:

q : $\Sigma N\ (\lambda x \rightarrow \text{if } x = 1 \text{ then } N \text{ else } \mathit{Bool})$

Werte für q:

q = 4, true

q = 1, 42

Dependent Types (3)

$\Rightarrow$ noch genauere Typüberpüfungen / Invarianten z.B.:

- Vektoren (auch mit GADTs)
- Matrizen ($m \times n \rightarrow n \times o \rightarrow m \times o$)
- Binäre Bäume (links $<$ mitte $<$ rechts)

$\approx$ Vermischung Typ/Wert

$\approx$ Endlosschleifen im Typsystem

$\approx$ Typgleichheit