Funktionale Programmierung
Kombinatorenbibliotheken und EDSL

1 DSL/EDSL

2 Abstraktion mit Applicative und Monad

- Abstraktionen am Beispiel der Fehlerbehandlung

Wieso (E)DSLs

- Fachleute Sprechen in den Worten ihrer Industrie
  - Versicherungsindustrie: Limite, Selbstbehalt, etc.
  - Handel: Call/Put, Option, Aktien etc.
  - Grafik: Vektor, Linie, Bézier Kurve, etc.
- Entwickler sprechen "Software"
  - Datentypen
  - Funktionen/Prozeduren
  - Module

- (Fast) Alle Industrien “wollen” mit Software effizienter und automatisierter werden.
- Viele Industrien sind (in dem Mass in dem sie heute betrieben werden) sogar nur im Kontext von Software sinnvoll (z.B. 3-d Grafik (Modellierung und Animation) elektronischer Handel, etc.)
- Die in der Industrie verwendeten Programme müssen die Sprache und Begriffe der Industrie als Inputs verstehen und verarbeiten können.
- Wenn die Industriefachleute ihre Automatisierung selber (mit)gestalten wollen, dann braucht es sogar spezielle Programmiersprachen, deren Keywords möglichst direkt Industriespezifische Begriffe und Konstrukte widerspiegeln und interpretieren.

Dies leisten

Domänenspezifische Sprachen / Domain Specific Languages (DSLs)

Man unterscheidet typischerweise zwischen *Embedded Domain Specific Languages* (EDSL) und “normalen” DSLs.

DSL:

- Ist eine eigenständige formale Sprache.
- Hat eigene Interpreter/Compiler, Parser, etc.

EDSL:

- Ist in einer bestehenden Programmiersprache (host language) eingebettet (embedded).
- Terme der EDSL sind auch Terme der host language (eine echte Teilmenge). Typen der host language stehen für Elemente der abgebildeten Domäne, Funktionen werden zum kombinieren und manipulieren dieser Elemente verwendet.

Haskell bietet eine geeignete Umgebung zum Implementieren von EDSLs.

- Ausdrucksstarkes Typensystem.
- Möglichkeit zur Abstraktion (z.B. höhere Funktionen, Typklassen).

Wir machen eine einfache Beispiel EDSL für 2D-Grafiken in Haskell (Sie haben mit Gloss bereits Erfahrungen mit einer ähnlichen EDSL/Kombinatorenbibliothek gemacht).

- Wir beginnen mit einigen Grundformen (e.g. Kreis, Quadrat, etc)
- Wir definieren Funktionen zum Kombinieren und Modifizieren von Formen.

Grundformen

```
1 -- | Basic Shapes
2 empty :: Shape
3
4 unitDisc :: Shape
5
6 unitSq :: Shape
```

Modifikatoren

```
1 translate :: Vector -> Shape -> Shape
2 stretchX  :: Float  -> Shape -> Shape
3 stretchY  :: Float  -> Shape -> Shape
4 stretch   :: Float  -> Shape -> Shape
5 flipX     :: Shape  -> Shape
6 flipY     :: Shape  -> Shape
7 flip45    :: Shape  -> Shape
8 flip0     :: Shape  -> Shape
9 rotate    :: Float  -> Shape -> Shape
```

Kombinatoren

```
1 intersect :: Shape -> Shape -> Shape
2
3 merge :: Shape -> Shape -> Shape
4
5 minus :: Shape -> Shape -> Shape
```

Jetzt haben wir bereits die Syntax (AST) unserer EDSL beschrieben. Wir können jetzt Formen direkt beschreiben:

```
1 iShape = merge
2     (stretchY 2 unitSq)
3     (translate (0, 4) unitDisc)
```

Wir wollen nun eine Semantik für unsere EDSL angeben. Die Semantik soll darin bestehen, dass wir Formen, die wir in unserer Sprache beschreiben als Bild wiedergeben können.

Wir halten es einfach (keine Farben etc.) und interpretieren eine “Shape” als Menge von Pixeln in der Ebene vie der charakteristischen Funktion:

```
1 type Point = (Float, Float)
2
3 newtype Shape = Shape { inside :: Point -> Bool
    }
```

Aufgabe

Implementieren Sie die einzelnen Kombinatoren und Grundformen.

Shallow embedding haben wir vorher für unsere “Shape” EDSL gemacht.
Grundformen plus Funktionen, die Formen verändern und kombinieren.

Bei einem "Deep Embedding" entspricht die EDSL einem Datentyp in Haskell:

```haskell
data Shape
    = Empty
    | UnitDisk
    | UnitSq
    | Translate Vector Shape
    | ...
```

Die Konstruktoren dieses Typs haben genau die gleiche Signatur wie die Funktionen in der ersten Implementierung.

Syntax und Semantik sind klarer getrennt in einem “Deep Embedding”. Wir mussten bei unserer Implementation zum Beispiel schon mit dem Typ

```
1 newtype Shape = Shape {inside :: Point -> Bool}
```

Anfangen. In einem Deep embedding wäre dies einfach Teil einer mögliche Semantik (diese Interpretation wäre dann ein einem Ort “konzentriert”).

```
1 inside :: Point -> Shape -> Bool
```

Interessantere EDSLs bestehen meist aus einer Mischung dieser beiden Ansätze (vgl. unsere EDSL für reguläre Ausdrücke).

Eine extrem einfache EDSL für arithmetische Ausdrücke:

```haskell
data Exp
    = Const Int
    | Add Exp Exp
    | Mul Exp Exp
    | Div Exp Exp
```

und eine ebensolche Interpretation:

```haskell
sEval :: Exp -> Int
sEval e = case e of
    Const x -> x
    Add e1 e2 -> sEval e1 + sEval e2
    Mul e1 e2 -> sEval e1 * sEval e2
    Div e1 e2 -> sEval e1 `div` sEval e2
```

Probleme?

```
1 x = Const 304
2 y = Const 7
3 z = Const 0
4 good = (x `Add` z) `Div` y
5 bad = (x `Add` y) `Div` z
```

```
1 *Main> sEval bad
2 *** Exception: divide by zero
3 *Main>
```

Wir wollen in unserer Interpretation ein eifaches Fehlermanagement berücksichtigen:

```haskell
data Result a
    = DivisionByZeroError
    | Success a
    deriving Show
```

Aufgabe

Implementieren Sie:

```haskell
evalE :: Exp -> Result Int
```

Viel “Handarbeit” beim Fehlerhandling:

```
1  evalE :: Exp -> Result Int
2  evalE e = case e of
3      Const x -> Success x
4      Div e1 e2 -> case evalE e1 of
5        DivisionByZeroError -> DivisionByZeroError
6        Success x1 -> case evalE e2 of
7          DivisionByZeroError ->
               DivisionByZeroError
8          Success x2
9            | x2 /= 0 -> Success $ x1 `div` x2
10           | otherwise -> DivisionByZeroError
11     Add e1 e2 -> case evalE e1 of
12       DivisionByZeroError -> DivisionByZeroError
13       Success x1 -> case evalE e2 of
14         DivisionByZeroError -> ...etc.
```

Die “Monad”, “Applicative” und “Functor” Typklassen helfen uns diese Handarbeit zu abstrahieren

Der “Result” Typ ist ein Funktor:

```haskell
instance Functor Result where
    fmap f (Success x) = Success $ f x
    fmap f _ = DivisionByZeroError
```

Verifizieren Sie die Funktor Regeln!

Der “Result” Typ ist sogar ein “Applicative” Funktor:

```haskell
instance Applicative Result where
    Success f <*> Success x = Success $ f x
    _ <*> _ = DivisionByZeroError
    pure = Success
```

Verifizieren Sie die Regeln!

Der “Result” Typ ist auch ein “Monad”:

```
1 instance Monad Result where
2     Success x >>= f = f x
3     _ >>= _ = DivisionByZeroError
```

Verifizieren Sie die Regeln!

Aufgabe

Implementieren Sie nochmals

```
1 evalE2 :: Exp -> Result Int
```

Verwenden Sie diesmal die Funktor, Monad und Applicative Instanzen.

Abstraktion beim Fehlerhandling:

```
1  eval2 :: Exp -> Result Int
2  eval2 e = case e of
3    Const x -> pure x
4    Add e1 e2 -> (+) <$> eval2 e1 <*> eval2 e2
5    Mul e1 e2 -> (*) <$> eval2 e1 <*> eval2 e2
6    Div e1 e2 -> do
7      x1 <- eval2 e1
8      x2 <- eval2 e2
9      if x2 == 0 then
10         DivisionByZeroError
11     else
12         return $ x1 `div`x2
```