Praktische Informatik 3: Funktionale Programmierung
Vorlesung 12 vom 15.01.2019: Domänenspezifische Sprachen (DSLs)

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- **Teil III: Funktionale Programmierung im richtigen Leben**
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Domain-Specific Languages (DSLs)

- Was ist das?
- Wie macht man das?
- Wozu braucht man so etwas?

Programmiersprachen sind überall

- Beispiel 1: **SQL** — Anfragesprache für relationale Datenbanken
- Beispiel 2: **Excel** — Modellierung von Berechnungen
- Beispiel 3: **HTML** oder **LaTeX** oder **Word** — Typesetting

Vom Allgemeinen zum Speziellen

- Modellierung von **Problemen** und **Lösungen**

Allgemein ⟷ Spezifisch

Allgemeine Lösung: GPL
- Mächtige Sprache (Turing-mächtig)
- Große Klasse von Problemen
- Großer Abstand zum Problem
- Java, Haskell, C ...
- General purpose language (GPL)

Spezifische Lösung: DSL
- Maßgeschneiderte Sprache
- Wohldefinierte Unterklasse (Domäne) von Problemen
- Geringer Abstand zum Problem
- Als Teil einer Programmiersprache (eingebettet) oder alleinstehend (stand-alone)

DSL: Definition 1

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

(van Deursen et al., 2000)

Eigenschaften von DSLs

- **Fokussierte** Ausdrucksmächtigkeit
  - Turing-Mächtigkeit nicht Ziel der Sprache (aber kein Ausschlusskriterium)
  - Oftmals deutlich weniger mächtig: Reguläre Ausdrücke, Makefiles, HTML
- Üblicherweise **klein** ("little languages", "micro-languages")
- Anzahl der Sprachkonstrukte eingeschränkt und auf die Anwendung zugeschnitten
- Meist **deklarativ**: XSLT, Relax NG Schemas, Excel Formeln...
- Spiegeln in Sprachkonstrukten und Vokabular die Domäne wider

Domain-Specific Embedded Languages

- DSL direkt in eine GPL **einbetten**
  - Vorhandenes Ausführungsmodell und Werkzeuge
- Funktionale Sprachen eignen sich hierfür besonders gut
  - Algebraische Datentypen zur Termrepräsentation
  - Funktional $\subseteq$ Deklarativ
  - Funktionen höherer Ordnung ideal für **Kombinatoren**
  - Interpreter (`ghci`, `ocaml`, ...) erlauben "rapid prototyping"
  - Erweiterung zu **stand-alone** leicht möglich
- Andere Sprachen:
  - Java: Eclipse Modelling Framework, Xtext

Beispiel: Reguläre Ausdrücke

Ein regulärer Ausdruck ist:

- Leeres Wort ϵ
- Einzelnes Zeichen c
- Beliebiges Zeichen ?
- Sequenzierung $e_1\ e_2$
- Alternierung $e_1 \mid e_2$
- Kleene-Stern $e^* = \epsilon \mid ee^*$
- Abgeleitet:
  - Kleene-Plus $e^+ = e\ e^*$

Haskell-Implementierung — Signatur:

```
type RegEx

eps   :: RegEx
char  :: Char→ RegEx
arb   :: RegEx
seq   :: RegEx→ RegEx→ RegEx
alt   :: RegEx→ RegEx→ RegEx
star  :: RegEx→ RegEx
```

Implementierung: siehe `SimpleRegEx.hs`

Flache Einbettung vs. Tiefe Einbettung

- **Flache Einbettung:**
  - Domänenfunktionen direkt als Haskell-Funktionen
  - Keine explizite Repräsentation der Domänenobjekte in Haskell
  - Schnell geschrieben und flexibel erweiterbar
- **Tiefe Einbettung:**
  - Repräsentation der Domänenobjekte durch Haskell-Datentyp (oder ADT)
  - Domänenfunktionen auf diesem Datentyp
  - Mächtiger: Manipulation der Domänenobjekte möglich (Reflektion)

Beispiel: Grafik

- Erzeugung von SVG-Grafiken
- Eingebettete DSL:
  - Monade Draw (Zustandsmonade)
  - Funktionen zum Zeichnen:

```
line    :: Point→ Point→ Draw ()
polygon :: [Point]→ Draw ()
```

  - "Ausführen": Darstellung in S.Svg, rendering:

```
render :: Int→ Int→ Draw()→ S.Svg
draw   :: String→ S.Svg → IO ()
```


Beispielprogramm: Sierpińksy-Dreieck

Dreieck mit Eckpunkten zeichnen:

```
drawTriangle :: Point→ Point→ Point→ Draw ()
```

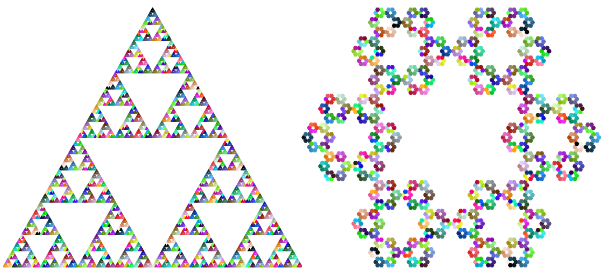
Mitte zwischen zwei Punkten:

```
midway :: Point→ Point→ Point
midway p q = 0.5 `smult` (p+q)
```

Sierpińksy-Dreieck rekursiv

```
spTri :: Double→ Int→ Draw ()
spTri sz limit = sp3 a b c 0 where
  h = sz* sqrt 3/4
  a = Pt 0 (−h); b = Pt (−sz/2) h; c= Pt (sz/2) h
  sp3 :: Point→ Point→ Point→ Int→ Draw ()
  sp3 a b c n
    | n ≥ limit = drawTriangle a b c
    | otherwise = do
        let ab = midway a b; bc = midway b c; ca = midway c a
        sp3 a ab ca (n+1); sp3 ab b bc (n+1); sp3 ca bc c (n+1)
```


Resultat: Sierpińsky-Dreieck und Schneeflocke



Erweiterung: Transformation

- Allgemein: **Transformation** von Grafiken

```
transform :: S.AttributeValue→ Draw()→ Draw()
```

- Speziell:
  - Rotation um einen Punkt:

```
rotate :: Point→ Double→ Draw ()→ Draw ()
```

  - Skalierung um einen Faktor:

```
scale :: Double→ Draw()→ Draw ()
```

  - Verschiebung um einen Vektor (Punkt):

```
translate :: Point → Draw () → Draw ()
```


Beispiele: Verschiebung und Skalierung

Implementierung

- Abbildung auf weitere DSL: Text.Blaze.SVG
- Industrie-taugliche Modellierung von Markup-Sprachen (SVG, HTML)
  - Kann mehr als String (ByteString, Text)
  - Skaliert auch für große Texte
- Umgeht Probleme:
  - String nicht effizient
  - Insbesondere Zeichenketten von vorne aufzubauen ist **ineffizient**
- Aber noch nahe an SVG/HTML: keine Typisierung, keine Ausdrucksmächtigkeit

Weitere Abgrenzung

Programmierschnittstellen (APIs)

- Etwa jUnit: `assertTrue()`, `assertEquals()` Methoden & `@Before`, `@Test`, `@After` Annotationen
- Funktionsnamen spiegeln ebenfalls Domänenvokabular wider
- Gängige Sprachen (Java, C/C++) erschweren weitere Abstraktion: Syntaxerweiterungen, Konzepte höherer Ordnung
- Imperative Programmiersprache vs. deklarative DSL

Skriptsprachen

- JavaScript, PHP, Lua, Tcl, Ruby werden für DS-artige Aufgaben verwendet
  - HTML/XML DOM-Manipulation
  - Game Scripting, GUIs, ...
  - Webprogrammierung (Ruby on Rails)
- Grundausrichtung: programmatische Erweiterung von Systemen

Beispiel: Hardware Description Languages

- Ziel: Funktionalität von Schaltkreisen beschreiben
- Einfachster Fall:

```
and :: Bool→ Bool→ Bool
or  :: Bool→ Bool→ Bool
```

- Moderne Schaltkreise sind etwas komplizierter ...

CλaSH

- Modellierung und Simulation von Schaltkreisen in Haskell
- Typ $\mathsf{Signal}\ \alpha$ für synchrone sequentielle Schaltkreise
- Rekursion für Feedback
- Simulation des Verhalten des Schaltkreises möglich
- Generiert VHDL, Verilog, SystemVerilog, und Testdaten

- Verwandt: Chisel (in Scala), Bluespec (kommerziell), Lava (veraltet)

Beispiel: SQL

- SQL-Anfragen werden in Haskell modelliert, dann übersetzt und an DB geschickt
- Vorteil: typsicher, ausdrucksstark
- Wie modelliert man das **Ergebnis**? → Abbildung Haskell-Typen auf DB
- Haskell: Squeal
- Scala: Slick

Vorteile der Verwendung von DSLs

- Ausdruck von Problemen/Lösungen in der Sprache und auf dem Abstraktionslevel der Anwendungsdomäne
- Notation matters: Programmiersprachen bieten oftmals nicht die Möglichkeit, Konstrukte der Domäne angemessen wiederzugeben
- DSL-Lösungen sind oftmals selbstdokumentierend und knapp
- Bessere (automatische) Analyse, Optimierung und Testfallgenerierung von Programmen
  - Klar umrissene Domänensemantik
  - eingeschränkte Sprachmächtigkeit ⇒ weniger Berechenbarkeitsfallen
- Leichter von Nicht-Programmierern zu erlernen als GPLs

Nachteile der Verwendung von DSLs

- Hohe initiale Entwicklungskosten
- Schulungsbedarf
- Sprachdesign ist eine äußerst schwierige und komplexe Angelegenheit, deren Aufwand nahezu immer unterschätzt wird
- Fehlender Tool-Support
  - Debugger
  - Generierung von (Online-)Dokumentation
  - Statische Analysen, ...
- Effizienz: Interpretation ggf. langsamer als direkte Implementierung in GPL

Zusammenfassung

- DSL: Maßgeschneiderte Sprache für wohldefinierten Problemkreis
- Vorteile: näher am Problem, näher an der Lösung
- Nachteile: Initialer Aufwand
- Klassifikation von DSLs:
  - Flache vs. tiefe Einbettung
  - Stand-alone vs. embedded
- Nächste Woche: Scala — eine Einführung.

Literatur

Koen Claessen and David Sands.
Observable sharing for functional circuit description.
In P. S. Thiagarajan and R. Yap, editors, *Advances in Computing Science – ASIAN'99*, volume 1742 of *LNCS*, pages 62–73, 1999.

Paul Hudak.
Building domain-specific embedded languages.
ACM Comput. Surv., 28, 1996.

Marjan Mernik, Jan Heering, and Anthony M. Sloane.
When and how to develop domain-specific languages.
ACM Comput. Surv., 37(4):316–344, 2005.

Arie van Deursen, Paul Klint, and Joost Visser.
Domain-specific languages: an annotated bibliography.
SIGPLAN Not., 35(6):26–36, 2000.