

Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 19.01.10:
Domain Specific Languages (DSLs)

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- ▶ Teil I: Monaden und fortgeschrittene Typen
- ▶ Teil II: Fortgeschrittene Datenstrukturen
- ▶ Teil III: Nebenläufigkeit
- ▶ Teil IV: The Future of Programming
 - ▶ Fancy Types
 - ▶ Domain-Specific Languages
 - ▶ The Next Big Thing: F#, Scala
 - ▶ Rückblick, Ausblick

Themen

- ▶ Domain specific languages: Definition & Einordnung
- ▶ Eingebettete DSLs in Haskell
 - ▶ Einbettung: Vor-/Nachteile
 - ▶ Kompilierung vs. Interpretation
 - ▶ Tiefe vs. flache Einbettung
 - ▶ Repräsentation von Graphen
- ▶ Beispiele
 - ▶ Statecharts
 - ▶ Haskell XML Toolkit
 - ▶ Lava Hardware Description Language

DSLs contra allgemeine Programmiersprache

- ▶ Bei (informatischen) Problemstellungen gibt es oft zwei Möglichkeiten
 - ▶ eine spezifische Lösung, die **das konkrete** Problem sehr effizient/elegant/kostensparend löst, aber nicht/schwer auf andere/ggf. ähnliche Probleme anwendbar ist
 - ▶ eine generische Lösung, die auch andere Probleme abdeckt/abdecken kann, aber mehr Anpassungsaufwand erfordert/weniger effizient bei der Lösung des vorliegenden Problems ist
- ▶ Diese Dichotomie spiegelt sich in der Unterscheidung zwischen **allgemeinen Programmiersprachen** (general purpose languages, GPL) und **domänenspezifischen Sprachen** (DSL) wider
 - ▶ Lösen einer großen Klasse von Problemen durch Bereitstellen einer Bücherei in einer (meist Turing-mächtigen) GPL
 - ▶ Lösen einer wohldefinierten Unterklasse von Problemen mit einer DSL

Allgemeine Programmiersprache

- ▶ Das Konzept der domänenspezifischen Ausrichtung trifft im Grunde auch auf klassische Programmiersprachen zu:
 - ▶ Cobol wurde für die Abbildung von Geschäftsprozessen entworfen
 - ▶ Fortran: numerische Berechnungen (“number crunching”)
 - ▶ Lisp: Symbolische Berechnungen (v. a. in der KI)
 - ▶ Erlang: Telecommunication Switching, fehlertolerante nebenläufige Systeme
- ▶ Aber: diese wurden über die Zeit zu allgemeinen Programmiersprachen erweitert
- ▶ Wiederum ein generelles Phänomen: Sprachen tendieren dazu, “fett” zu werden.
 - ▶ Keine Sprache ist zur Entwurfszeit perfekt
 - ▶ Anwender fordern neue Features, die sie aus anderen Sprachen kennen
 - ▶ Neuentwurf/Reduktion einer Sprache nicht praktikabel, daher meist monotones Wachstum der bestehenden Sprache

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)

DSL: Definition 2

In software development, a domain-specific language (DSL) is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. The concept isn't new – special-purpose programming languages and all kinds of modeling/specification languages have always existed –, but the term has become more popular due to the rise of domain-specific modeling.

(Wikipedia, 2010-01-18)

Eigenschaften von DSLs

- ▶ Fokussierte Ausdrucksmächtigkeit
 - ▶ Turing-Mächtigkeit kein Ausschlusskriterium, aber auch nicht Ziel der Sprache.
 - ▶ Oftmals deutlich weniger mächtig: Reguläre Ausdrücke, Makefiles,
- ▶ DSLs sind üblicherweise klein (daher auch die Bezeichnungen “little language” und “micro-language”), d. h. die Anzahl der Sprachkonstrukte ist eingeschränkt, aber auf die Anwendung zugeschnitten
- ▶ Meist sind sie deklarativ statt imperativ: XSLT, Relax NG Schemas, Graphviz/Dot, Excel Formeln. . .
- ▶ Spiegeln in Sprachkonstrukten und Vokabular die Domäne wider

Weitere Abgrenzung

Programmierschnittstellen (APIs)

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

Weitere Abgrenzung

Programmierschnittstellen (APIs)

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

Skriptsprachen

- ▶ JavaScript, Lua, Ruby werden für DS-artige Aufgaben verwendet
 - ▶ HTML/XML DOM-Manipulation
 - ▶ Game Scripting (z. B. mit Lua in World of Warcraft)
 - ▶ Webprogrammierung: Ruby on Rails
- ▶ Grundausrichtung: programmatische Erweiterung von Systemen

DSL-Beispiel: Relax NG

Adressbuchformat

```
grammar {  
  start = entries  
  entries = element entries { entry* }  
  entry = element entry {  
    attribute name { text },  
    attribute birth { xsd:dateTime },  
    text }  
}
```

- ▶ Beschreibung von XML-Bäumen
 - ▶ Erlaubte Element-Verschachtelungen & -Reihenfolgen
 - ▶ Datentypen von Attributen & Elementwerten
- ▶ Mögliche automatische Generierung von Validatoren

DSL-Beispiel: Reguläre Ausdrücke

Textsuche und -ersetzung

```
sed -e 's/href=\(\("[^"]*"*\)\)/src=\1/g'
```

```
egrep '^\[.*\]$'
```

- ▶ Extrem effiziente Implementierung als endliche Automaten
- ▶ Konzise, wenn auch unleserliche, Beschreibung von Textmustern
- ▶ Elementarer Bestandteil von Skriptsprachen (Perl, Python, JavaScript...)

DSL-Beispiel: VHDL

```
ENTITY DFlipflop IS
    PORT(D,Clk: IN Bit;
         Q: OUT Bit);
END DFlipflop;
ARCHITECTURE Behav OF DFlipflop IS
    CONSTANT T_Clk_Q: time := 4.23 ns;
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clk'EVENT AND Clk'Last_Value='0' AND
                Clk='1';

        Q<=D AFTER T_Clk_Q;
    END PROCESS;
END Behav;
```

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 \Rightarrow 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

Auswertung von DSLs

- ▶ Kompilierung: Maßgeschneiderter Code für ein gegebenes DSL-Programm wird erzeugt (und ggf. übersetzt)
 - ▶ *Lex/Yacc* erzeugen C Code für spezifischen Lexer (insb. Zustandsautomat) und Parser
 - ▶ *Pan* (Functional Images, C. Elliott) übersetzt Haskell Datentypen in C Programme, die Bilddateien bzw. Bildanzeige generieren
- ▶ Interpretation mit tiefer Einbettung
 - ▶ Programme als Datentypen (`Expr`) der interpretierenden Sprache (host language)
 - ▶ Interpretationsfunktion $I : Expr \rightarrow Env \rightarrow Value$ evaluiert Programm und Eingaben zu Wert der Domäne (+ Ausgaben)
- ▶ Interpretation mit flacher Einbettung
 - ▶ DSL-Operatoren arbeiten direkt auf semantischer Domäne
 - ▶ `Add : Value -> Value -> Value`

Domain-specific embedded languages

- ▶ Um Probleme mit Sprachdefinition und Interpreterimplementierung zu vermeiden, kann eine DSL auch direkt in eine GPL eingebettet werden
 - ▶ 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 mit sich entwickelnder DSL herumzuspielen

XML-Verarbeitung mit dem Haskell XML Toolkit (HXT)

- ▶ Eine in Haskell eingebettete DSL zur Verarbeitung von XML-Dokumenten
 - ▶ Eingabe/Ausgabe von XML
 - ▶ Transformationen auf XML DOM
- ▶ Kann als XPath-Ersatz dienen (zur Auswahl von Bestandteilen eines XML-Dokuments); durch Einsatz von beliebigen Haskell-Code jedoch deutlich mächtiger
- ▶ Ebenfalls möglich: Einsatz als XSLT-Ersatz: Transformation von einem XML-Schema in ein anderes
- ▶ Kombinator-basiert: ein fester Satz von unären und binären Haskell-Funktionen repräsentiert die Sprache zur Beschreibung von XML *Filtern*

HXT: Datentypen für XML-Dokumente

```
data NTree a = NTree a [NTree a]  — rose tree

data XNode   = XText String  — plain text node
              | ...
              | XTag QName XmlTrees
                — element name and list of attributes
              | XAttr QName   — attribute name
              | ...

type QName   = ...  — qualified name

type XmlTree = NTree XNode
```

HXT: XML Filter

- ▶ Elementare Datenstruktur: Mehrwertige Funktion über XmlTree

```
type XmlFilter = XmlTree → [XmlTree]
```

```
type Filter a b = a → [b]
```

```
isA    :: (a → Bool) → (a → [a])
```

```
isA p x
```

```
  | p x      = [x]
```

```
  | otherwise = []
```

```
isXText      :: XmlFilter
```

```
isXText t@(NTree (XText _) _) = [t]
```

```
isXText _     = []
```

HXT: Transformationen & partielle Fkt.

```
trans    :: XmlTree → XmlTree
```

```
trans t = exp(t)
```

```
ftrans   :: XmlTree → [XmlTree]
```

```
ftrans t = [exp(t)]
```

```
part     :: XmlTree → XmlTree
```

```
part t
```

```
  | p t = expr(t)
```

```
  | otherwise = error "f_ not_ defined"
```

```
fpart    :: XmlFilter
```

```
fpart t
```

```
  | p t = [expr(t)]
```

```
  | otherwise = []
```

HXT: Komposition von Filtern

- ▶ Zwei Filter nacheinander ausführen:

```
(>>>)      :: XmlFilter → XmlFilter → XmlFilter  
(f >>> g) t = concat [g t' | t' ← f t]
```

- ▶ Ergebnisse zweier Filter zusammentun:

```
(<+>)      :: XmlFilter → XmlFilter → XmlFilter  
(f <+> g) t = f t ++ g t
```

HXT: Auswahl

```
orElse :: XmlFilter → XmlFilter → XmlFilter
```

```
orElse f g t
```

```
| null res1 = g t
```

```
| otherwise = res1
```

```
where res1 = f t
```

```
when :: XmlFilter → XmlFilter → XmlFilter
```

```
when f g t
```

```
| null (g t) = [t]
```

```
| otherwise = f t
```

```
guards :: XmlFilter → XmlFilter → XmlFilter
```

```
guards g f t
```

```
| null (g t) = []
```

```
| otherwise = f t
```

HXT: Traversal des XML-Baumes

- ▶ Auswahl aller Nachfahren, die f erfüllen, wobei Kinder von derlei Elementen nicht weiter untersucht werden

```
deep    :: XmlFilter → XmlFilter
deep f  = f 'orElse' (getChildren >>> deep f)
```

- ▶ Auswahl wirklich aller Nachfahren, die f erfüllen

```
multi   :: XmlFilter → XmlFilter
multi f = f <+> (getChildren >>> multi f)
```

- ▶ Weitere Beispiele: `examples-11/Hxt.hs`

HXT: Das Kleingedruckte

- ▶ HXT baut als Bücherei auf dem Konzept der *Arrows* auf
- ▶ Siehe <http://www.haskell.org/arrows/>

```
class Category cat where  
  id :: cat a a  
  (.) :: cat b c → cat a b → cat a c  
(<<<<) :: Category c ⇒ c b d → c a b → c a d  
(>>>) :: Category c ⇒ c a b → c b d → c a d
```

```
class Category a ⇒ Arrow a where  
arr :: (b → c) → a b c  
first :: a b c → a (b, d) (c, d)  
second :: a b c → a (d, b) (d, c)  
(***) :: a b c → a b' c' → a (b, b') (c, c')  
(&&&) :: a b c → a b c' → a b (c, c')
```

And now for something completely different. . .

Lava: Hardware Description Language

- ▶ In Haskell eingebettete Sprache zur Beschreibung von Schaltkreisen
- ▶ Industrieller Einsatz: Xilinx Inc. (S. Singh & P. Bjesse)
- ▶ In Universität Chalmers, Göteborg zur Lehre von Design und Verifikation von Hardware eingesetzt
- ▶ Dokumentation:
`http://www.cs.chalmers.se/~koen/Lava/tutorial.ps`
- ▶ Wir bauen die wichtigsten Bestandteile hier und jetzt *live* nach!
- ▶ Siehe `examples-11/Lava.hs`

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.