

Extensible domain specific languages

Christoph Schmalhofer Alexander Biehl

23. März 2016

Example DSL: Car configuration

Domain specific language from supplier

Hersteller

Modell

Typ

Farbe / Polster

Ausstattung








Laufzeit

Fahrzeug anfordern

Farben/Polster

Weiter

Bitte wählen Sie Ihre Ausstattung:

	300	Alpinweiß (Weiß)	0,00 €	+
	B45	Esti	0,00 €	+
	A83	Gla	0,00 €	+
	LC	Led	68,07 €	+
	LCL5	Led Inte Esti	0,00 €	+
	LCCX	Led	0,00 €	+
	LCSW	Led	0,00 €	+


Außenfarben: weiß
erfordert:
[LCL5] Leder Dakota Korallrot/Akzent Schwarz - nicht mit
Interieurleiste Aluminium Hexagon mit Akzentleiste Estorilblau
matt beim M Sport
oder
[LCCX] Leder Dakota Oyster
oder
[LCSW] Leder Dakota Schwarz
oder
[HAAT] Stoff/Alcantara Hexagon Anthrazit/Akzent Blau
nicht in Verbindung mit:
[met] Metallic-Lackierung


Ihre Auswahl:


Hersteller:
BMW


Modell:
1er-Reihe

Typ:
M135i 3.0 240 kW
3T Fli





 [Fahrzeug parken](#)

 [Fahrzeug vergleichen](#)

Preis

Example DSL: Car configuration









Domain specific language from supplier

Equipment	Condition	Rule
5	NOT 2	(6 OR 7) AND (8 OR 9)

For equipment 5

If condition 2 *isn't* met, 6 *or* 7 *and* 8 *or* 9 must be selected.

Partial evaluation of inclusions and logic solver for exclusions

Farben/Polster							Preis
Bitte lösen Sie zunächst die Abhängigkeiten auf:							
			LC	Leder Dakota		1.336,13 €	-
			LCL5	Leder Dakota Korallrot/Akzent Schwarz - nicht mit Interieurleiste Aluminium Hexagon mit Akzentleiste Estorilblau matt beim M Sport		0,00 €	+
			LCCX	Leder Dakota Oyster		0,00 €	+
			LCSW	Leder Dakota Schwarz		0,00 €	+
			LCL3	Leder Dakota Schwarz/Akzent Rot		0,00 €	+
			494	Sitzheizung für Fahrer und Beifahrer		277,31 €	+

Partial evaluation of inclusions and logic solver for exlusions

Farben/Polster

Bitte wählen Sie Ihre Ausstattung:

			300	Alpinweiß (Weiß)		0,00 €	
			uni	Alpinweiß Uni-Lackierung (Weiß)		210,08 €	
			B45	Estoril Blau Metallic (Blau)		0,00 €	
			A83	Glaciersilber Metallic (Silber)		0,00 €	
			A61	Karmesinrot (Rot)		0,00 €	
			LC	Leder Dakota		1.336,13 €	
			LCL5	Leder Dakota Korallrot/Akzent Schwarz - nicht mit Interieurleiste Aluminium Hexagon mit Akzentleiste Estorilblau		0,00 €	

Ihre Auswahl:

			494	Sitzheizung für Fahrer und Beifahrer		277,31 €	
--	--	--	-----	--------------------------------------	--	----------	--











Partial evaluation of inclusions and logic solver for exclusions

Hersteller Modell Typ **Farbe / Polster** Ausstattung Laufzeit Finanzierung anfordern

Farben/Polster

◀ ◀ ◀ ◀ ◀ ◀ 🔍 ⬆ Preis

Bitte lösen Sie zunächst die Abhängigkeiten auf:

			LC	Leder Dakota		1.336,13 €	−
			LCL5	Leder Dakota Korallrot/Akzent Schwarz - nicht mit Interieurleiste Aluminium Hexagon mit Akzentleiste Estorilblau		0,00 €	+
			LCCX	Leder Dakota Oyster		0,00 €	+
			LCSW	Leder Dakota Schwarz		0,00 €	+
			LCL3	Leder Dakota Schwarz/Akzent Rot		0,00 €	+

Scala Example: Shallow parsing of expressions

```
//Example: "NOT (1007) OR ( NOT ( {1008} OR {1009} OR {1010} OR {1011} OR {1012} OR {1013} OR {1014} OR {1015} ) )"

private lazy val atom = """[a-z0-9]+""".r ^^ { case x => Sym(x) }

private lazy val parens: Parser[Prop] = "(" ~> exp <~ ")"

private lazy val term: Parser[Prop] = parens | atom | not

private lazy val andterm = and | term

private lazy val or: Parser[Prop] = andterm ~ repl("OR" ~ andterm) ^^ {
  case f1 ~ fs => {
    val ot = new HashSet[Prop]();
    ot.add(f1);
    fs.map(p => { ot.add(p._2) })
    new Or(ot.toSet)
  }
}

private lazy val not: Parser[Prop] = ("NOT" | "-") ~ term ^^ { case f1 ~ f2 => { Not(f2) } };

private lazy val and: Parser[Prop] = term ~ repl("AND" ~ term) ^^ {
  case f1 ~ fs => {
    val ot = new HashSet[Prop]();
    ot.add(f1);
    fs.map(p => { ot.add(p._2) })
    new And(ot.toSet)
  }
}

private lazy val exp = (or | andterm)
```

Embedding a DSL

Choose an encoding

Shallow embedding

```
prop_1 = or (rule 2) (or (rule 6) (rule 7))
```

- ▶ Use functions to interpret terms directly.
- ▶ Fixed interpretation.

Example: Shallow embedding

```
type Proposition = ...
```

```
rule :: Int -> Proposition
```

```
rule = ...
```

```
or e1 e2 = e1 || e2
```

```
and e1 e2 = e1 && e2
```

Embedding a DSL

Choose an encoding

Deep embedding

```
prop_1 = Or (Rule 2) (Or (Rule 6) (Rule 7))
```

- ▶ Representation as abstract syntax tree.
- ▶ Interpret the AST with different interpreters.
- ▶ Roughly corresponds to visitor in OOP.

Example: Deep embedding

```
data Proposition = Rule Int
                | Or Proposition Proposition
                | And Proposition Proposition
                | ...
```

- ▶ scalac CNF transformation example: [http://www.scala-lang.org/api/2.11.2/scala-compiler/index.html#scala.tools.nsc.transform.patmat.Solving\\$CNF](http://www.scala-lang.org/api/2.11.2/scala-compiler/index.html#scala.tools.nsc.transform.patmat.Solving$CNF)

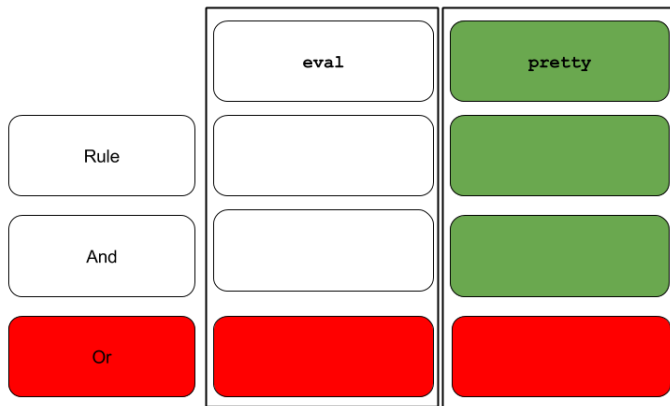
Expression problem

The Expression Problem
Philip Wadler, 12 November 1998

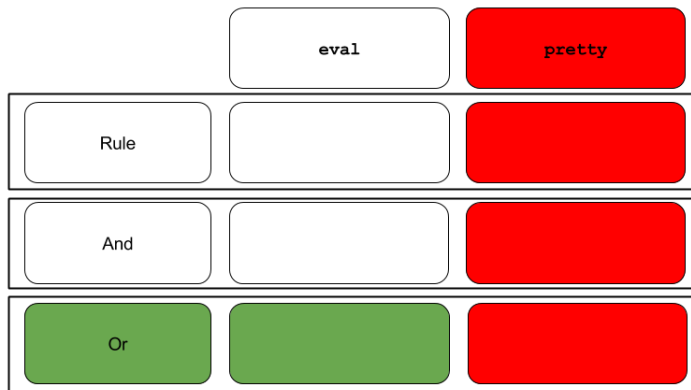
The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take expressions as the data type, begin with one case (constants) and one function (evaluators), then add one more construct (plus) and one more function (conversion to a string).

Abbildung : <http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt>

Modularity of deep embeddings



BTW: The dual in classical inheritance based OOP languages



Solving the expression problem

Final embedding

```
prop_1 = or (rule 2) (or (rule 6) (rule 7))
```

- Expressions as function term capturing denotational semantics.

Solving the expression problem

Final embedding

```
prop_1 :: Semantics repr => repr  
prop_1 = or (rule 2) (or (rule 6) (rule 7))
```

- Expressions as function term capturing denotational semantics.

Example: Finally tagless

```
class Semantics repr where
  true  :: repr
  false :: repr
  and   :: repr -> repr -> repr
  or    :: repr -> repr -> repr

newtype Eval = Eval { runEval :: Bool }

instance Semantics Eval where
  true  = Eval True
  false = Eval False
  and e1 e2 = runEval e1 && runEval e2
  or  e1 e2 = runEval e1 || runEval e2
```

Demo: Extension!

DEMO

Tagless final

- ▶ Polymorphic denotational semantic
 - ▶ Type safe
 - ▶ Concrete interpreter instantiated through type class mechanism
 - ▶ Modularity through Haskell's open type classes
 - ▶ 2-step transformations (reify/reflect)
 - ▶ Object sharing is explicit
-
- ▶ Polymorphic values can be hard to understand
 - ▶ Deserialization is tricky