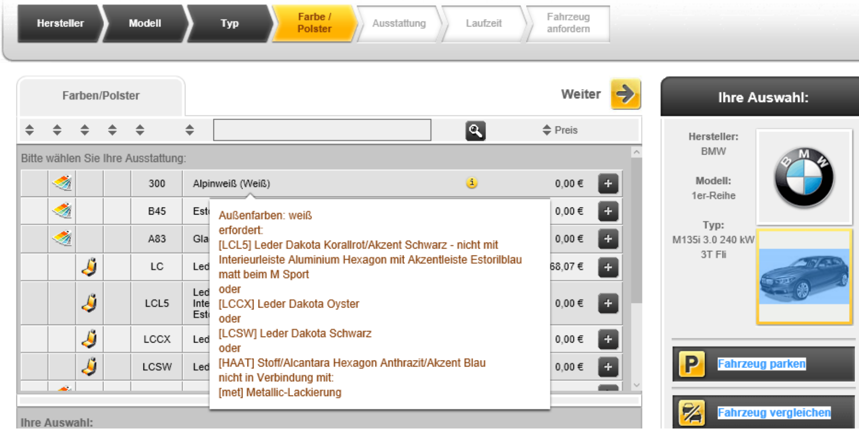
Extensible domain specific languages

Christoph Schmalhofer   Alexander Biehl

23. März 2016

Example DSL: Car configuration

Domain specific language from supplier



Example DSL: Car configuration

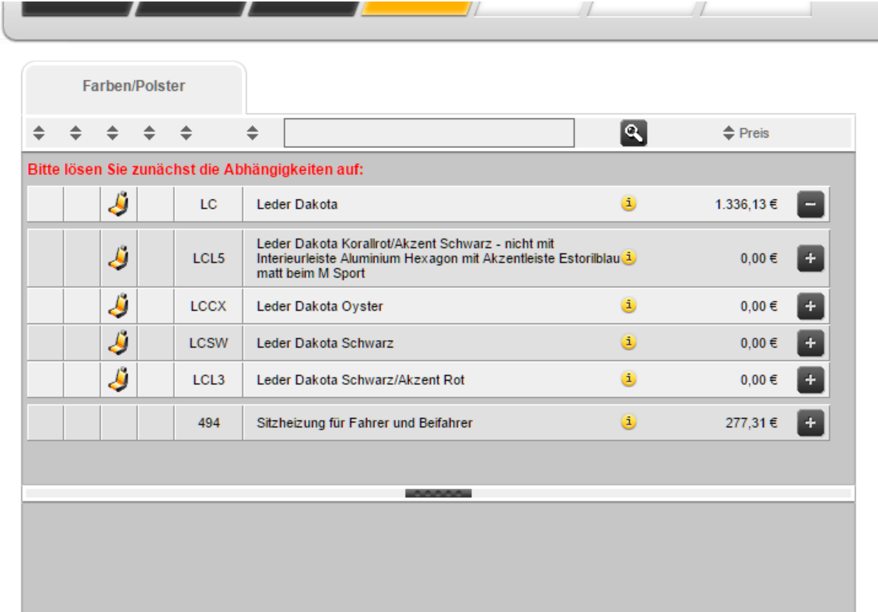
Domain specific language from supplier

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

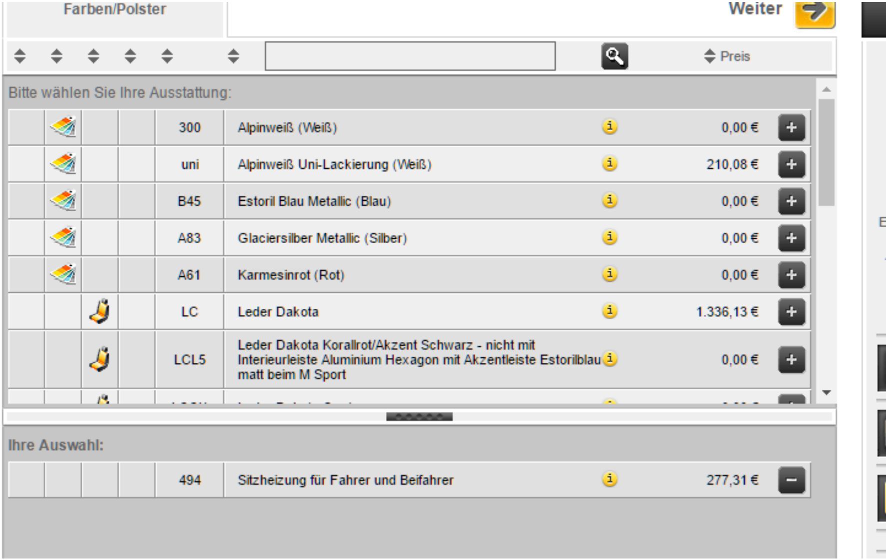
For equipment 5

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

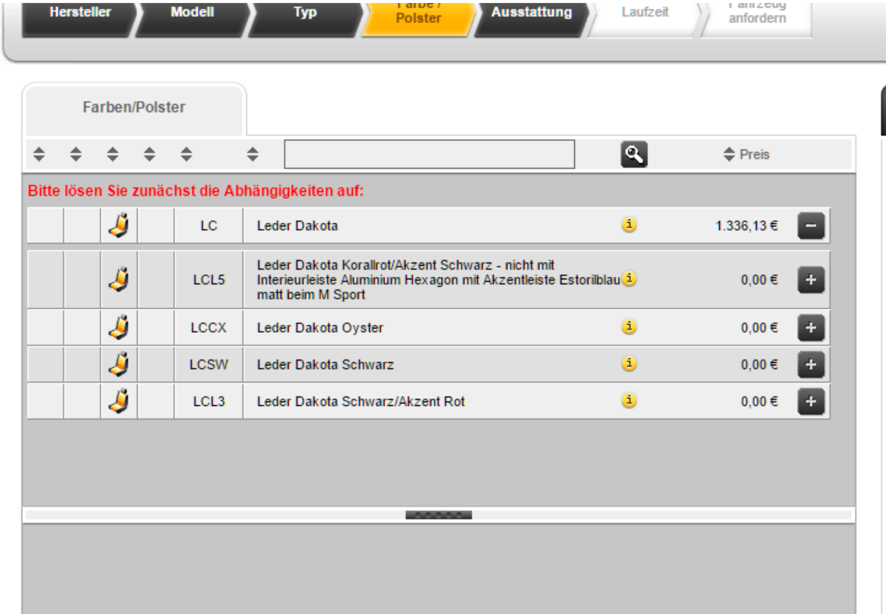
Partial evaluation of inclusions and logic solver for exlusions



Partial evaluation of inclusions and logic solver for exlusions



Partial evaluation of inclusions and logic solver for exlusions



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 ~ rep1("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 ~ rep1("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

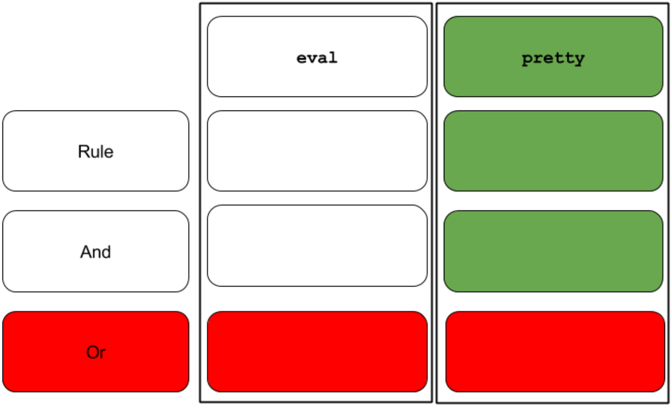
Expression problem

The Expression Problem
Philip Wadler, 12 November 1998

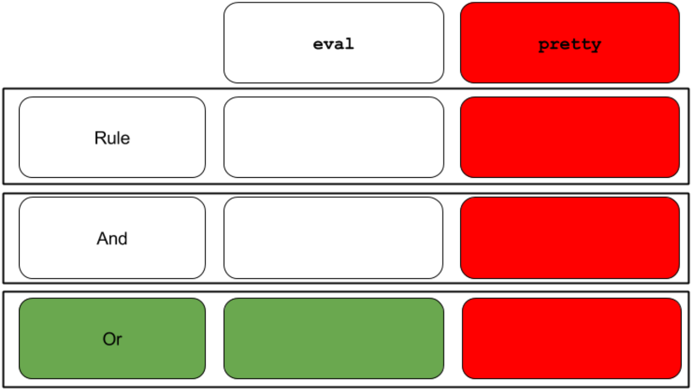
The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we 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 instanciated through type class mechanism
- Modularity through Haskells open type classes
- 2-step transformations (reify/reflect)
- Object sharing is explicit

- Polymorphic values can be hard to understand
- Deserialization is tricky