interactive creation of well-typed expressions in Domain Specific Languages

Pieter Koopman\textsuperscript{1}, Steffen Michels\textsuperscript{2}, Rinus Plasmeijer\textsuperscript{1,2}

1: Radboud University Nijmegen, 2: top-software.com

Top Software Technology

Radboud University
our use case: a DSL to query ships

- **AIS**
  - Automatic Identification System
  - transceiver reports on ship position, name, speed, kind, ..
  - typically a few times per minute

- **users need dynamic queries**
  - about ships, ports, wind-parks
  - distance, names, flags
  - ...

- **example**
  - which Dutch tankers are within 5 miles of a wind-park

screenshot from www.vesselfinder.com by OpenStreetMap

Radboud University
DSL design

• Grammar

Gen
 ::=  Ship Identifier Gen
    |  WindPark Identifier Gen
    |  Cond Cond Gen
    |  Return Expr

:: Cond
 ::=  LE Expr Expr
    |  And Cond Cond
    |  HasFlag Expr Country-list
    |  HasName Expr Name-list

:: Expr
 ::=  Identifier
    |  Number
    |  Distance Expr Expr
    |  Name Expr

• typical example

Ship s
  Cond (HasFlag s [NL])
WindPark p
  Cond (LE (Distance s p) 5)
  Return (Name s)

• the names of all Dutch ships that are within 5 miles of a windpark

• users need guidance to write queries
  • nautical experts, no computer scientists

• web-editor provides guidance

• strong typing prevents errors
DSL implementation: basic approach

- Grammar
  
  \[
  \text{Gen} \quad ::= \quad \text{Ship Identifier Gen} \\
  \quad \mid \quad \text{WindPark Identifier Gen} \\
  \quad \mid \quad \text{Cond Cond Gen} \\
  \quad \mid \quad \text{Return Expr}
  \]

  \[
  \text{Cond} \quad ::= \quad \text{LE Expr Expr} \\
  \quad \mid \quad \text{And Cond Cond} \\
  \quad \mid \quad \text{HasFlag Expr Country-list} \\
  \quad \mid \quad \text{HasName Expr Name-list}
  \]

  \[
  \text{Expr} \quad ::= \quad \text{Identifier} \\
  \quad \mid \quad \text{Number} \\
  \quad \mid \quad \text{Distance Expr Expr} \\
  \quad \mid \quad \text{Name Expr}
  \]

- data types matching the grammar

  \[
  \text{Gen} \quad ::= \quad \text{Ship Name Gen} \\
  \quad \mid \quad \text{WindPark Name Gen} \\
  \quad \mid \quad \text{Cond Cond Gen} \\
  \quad \mid \quad \text{Ret Expr}
  \]

  \[
  \text{Cond} \quad ::= \quad \text{LE Expr Expr} \\
  \quad \mid \quad \text{And Cond Cond} \\
  \quad \mid \quad \text{HasFlag Expr [Country]} \\
  \quad \mid \quad \text{HasName Expr [Name]}
  \]

  \[
  \text{Expr} \quad ::= \quad \text{Var Name} \\
  \quad \mid \quad \text{Num Real} \\
  \quad \mid \quad \text{Distance Expr Expr} \\
  \quad \mid \quad \text{Name Expr}
  \]
web-editor in iTasks

- Task Oriented Programming
- generate web-editors based on types

**derive class** iTasks Gen, Cond, Res

genTask :: Task Gen
genTask =
  withShared e0 \ g.
  updateSharedInformation [] \ g
  -|| viewSharedInformation [ViewAs run] \ g

Start w = doTasks genTask w
how this works

Ship "s" (Ret (Name (Var "s")))

RIGHT (CONS (LEFT (RIGHT CONS ...

user defined types

datatype-generic version

HTML + WA

GUI: web-editor

itasks

task-oriented programming framework

Clean

HTML

WA

Radboud University
limitations of this approach

• datatype generic programming
  ▪ one algorithm that work for 'any' datatype
  ▪ datatype must be known completely
  ▪ no functions, no quantifiers, no class restrictions

• datatypes allows ill-typed DSL expressions
  ▪ correct as datatype in host language
  ▪ no meaning in our DSL

HasFlag (Num 42.0) [NL]
Distance (Num 7.0) (String "Hello World!")
LE (Var "a") (Num 137.0)
Var "undefined name"

cannot be prevented by design of datatypes if we need overloading
making type-safe DSL-editors

1. type-checker for DSL
   - checks expression after user made it

2. DSL with phantom types
   - extra type parameter tells type of expression

3. DSL as GADT
   - Generalised Algebraic DataType

4. shallow embedded DSL
   - set of functions instead of datatype

5. let programmer define editor cases
   - this can assure correct types
   - phantom types, GADT, or functions

---

- too late: prevent errors
- too complex for datatype generic programming
- iTasks cannot derive this
- implemented here
dynamic editors

- programmer specifies a list of edit clauses
- system selects the options applicable

```haskell
dynamicEditor =
    [de "Add" \x y -> x + y :: Int Int -> Int,
     de "Eq" \x y -> x == y :: Int Int -> Bool,
     de "And" \x y -> x && y :: Bool Bool -> Bool,
     ce "Int.Val" intEditor,
     ce "Bool.Val" boolEditor]
```

rejected by the type system
list elements should have the same type
dynamic editors

- programmer specifies a list of edit clauses
- system selects the options applicable

\[
dynamicEditor = \\
[ \text{de "Add" (dynamic } \lambda x \ y \rightarrow x + y :: \text{Int Int} \rightarrow \text{Int)} \\
, \text{de "Eq" (dynamic } \lambda x \ y \rightarrow x == y :: \text{Int Int} \rightarrow \text{Bool)} \\
, \text{de "And" (dynamic } \lambda x \ y \rightarrow x && y :: \text{Bool Bool} \rightarrow \text{Bool)} \\
, \text{ce "Int.Val" intEditor} \\
, \text{ce "Bool.Val" boolEditor} \\
]
\]

\[
\text{intEditor :: Editor Int (?!Int)} \\
\text{intEditor = gEditor{\text{"\*\*\"}} EditValue}
\]

- transforms any type to Dynamic
- use editor recursively for arguments
- for a Bool result
**dynamic editors**

- programmer specifies a list of edit clauses
- system selects the options applicable

```haskell
dynamicEditor = 
  [de "Add" (dynamic \x y -> x + y :: Int Int -> Int)
   ,de "Eq" (dynamic \x y -> x == y :: Int Int -> Bool)
   ,de "And" (dynamic \x y -> x && y :: Bool Bool -> Bool)
   ,ce "Int.Val" intEditor
   ,ce "Bool.Val" boolEditor
  ]

intEditor :: Editor Int (?Int)
intEditor = gEditor{|*|} EditValue
```

for an integer result
type-safe editor for ship queries

dynamicEditor =
[  de "Has flag"
  (dynamic \ship list = \s->pure (isMember \s.flag list)
   :: (Eval Ship) [Country] -> Eval Bool)
,  de "Has name"
  (dynamic \x l = \x->pure (or (map (matchName \s.name) l))
   :: (Eval Ship) [String] -> Eval Bool)
,  de "And"
  (dynamic \x y = \x->y->pure (a && b)
   :: (Eval Bool) (Eval Bool) -> Eval Bool)
...

types are Monadic but very similar
type-safe variables

:: Expr = Var String | Not Expr | ..

• two problems
  1. variables carry no type, they can cause type errors
  2. variables can be undefined Var "x", any string will do

this is context sensitive!
type-safe variables

1. add a phantom type
   \[:: \text{VarName} \text{a} = \text{VarName String}\]

2. select variables from a pool of defined variables
   \[:: \text{Bind} = \{\text{idnt}::\text{String}, \text{val}::\text{Dynamic}\}\]

   \text{state0} :: [\text{Bind}]

   \text{state0} = [\{\text{idnt} = "s", \text{val} = \text{dynamic ship0}\}, \{\text{idnt} = "w", \text{val} = \text{dynamic windpark0}\}, ..]
using these variables

dynamicEditor =
[ de "Has flag"
    (dynamic \ship list = ship >>= \s. pure (isMember s.flag list)
      :: (Eval Ship) [Country] -> Eval Bool)
    : map toFunctionConsDyn state
.. 

toName :: Bind -> DynamicCons
toName {idnt, val = x :: t}
    = de idnt (dynamic (VarName idnt) :: VarName t)

match dynamic of type t
obtained query DSL for ships
operator overloading

• we need to indicate the dictionaries for overloading
• e.g. add them in with the Eq editor

\[
\text{de } \text{"Eq"}
\]

\[
(d\text{ynamic } \lambda (\text{EQ } f) \times y = f \times \gg= \lambda a. y \gg= \lambda b. \text{pure } (a == b)
\]

\[
:: A. a: (\text{EQ } a) \text{ (Eval } a\text{) (Eval } a\text{) } \rightarrow \text{ Eval } \text{Bool}
\]

, de "Number" (d\text{ynamic EQ id :: EQ } \text{Real}) ..

\[
:: \text{EQ } a = \text{EQ } ((\text{Eval } a)\rightarrow\text{Eval } a) \& == a
\]
overloading errors

- cannot prevent overloading errors during editing
- dynamic editors signal them immediately

\[
(d\text{ynamic} \ (EQ \ f) \ x \ y = f \ x \ >>= \ \a.\ y \ >>= \ \b.\pure \ (a == b) \\
:: A.\ a: \ (EQ \ a) \ (Eval \ a) \ (Eval \ a) \to \ Eval \ Bool)
\]

Please note the diagram showing fields filled in any order, and a message indicating a conflict in unifying Eval String with Eval Real.
conclusion

- we want type-safe editors for DSL expressions with guidance
- deriving editors for ADTs in iTask is wonderful
  - but often not good enough for a DSL (type errors, unsafe variables)

- dynamic-editors solve our problem
  - programmer specifies list of edit clauses
  - dynamic editor selects elements based on desired type
    - solves our problem: makes type-safe context sensitive editors
    - works for phantom types, GADTs and functions
    - can guarantee well-typed and defined variables
      - more work than just derive class iTask MyDSL
future work

• define variables on-the-fly:
  the new changing state in the editors allows this

• more flexible editors:
  change an expression \( x \) in \( \text{Add} \ x \ldots \)

• composable editors
  instead of one list of items for every editor

• …