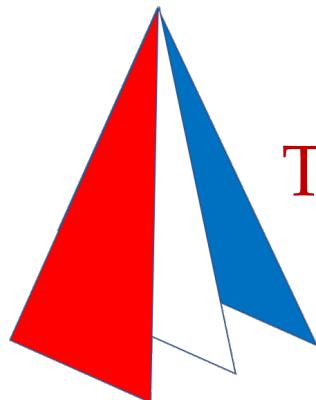


interactive creation of well-typed expressions in Domain Specific Languages

Pieter Koopman¹, Steffen Michels², Rinus Plasmeijer^{1,2}

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

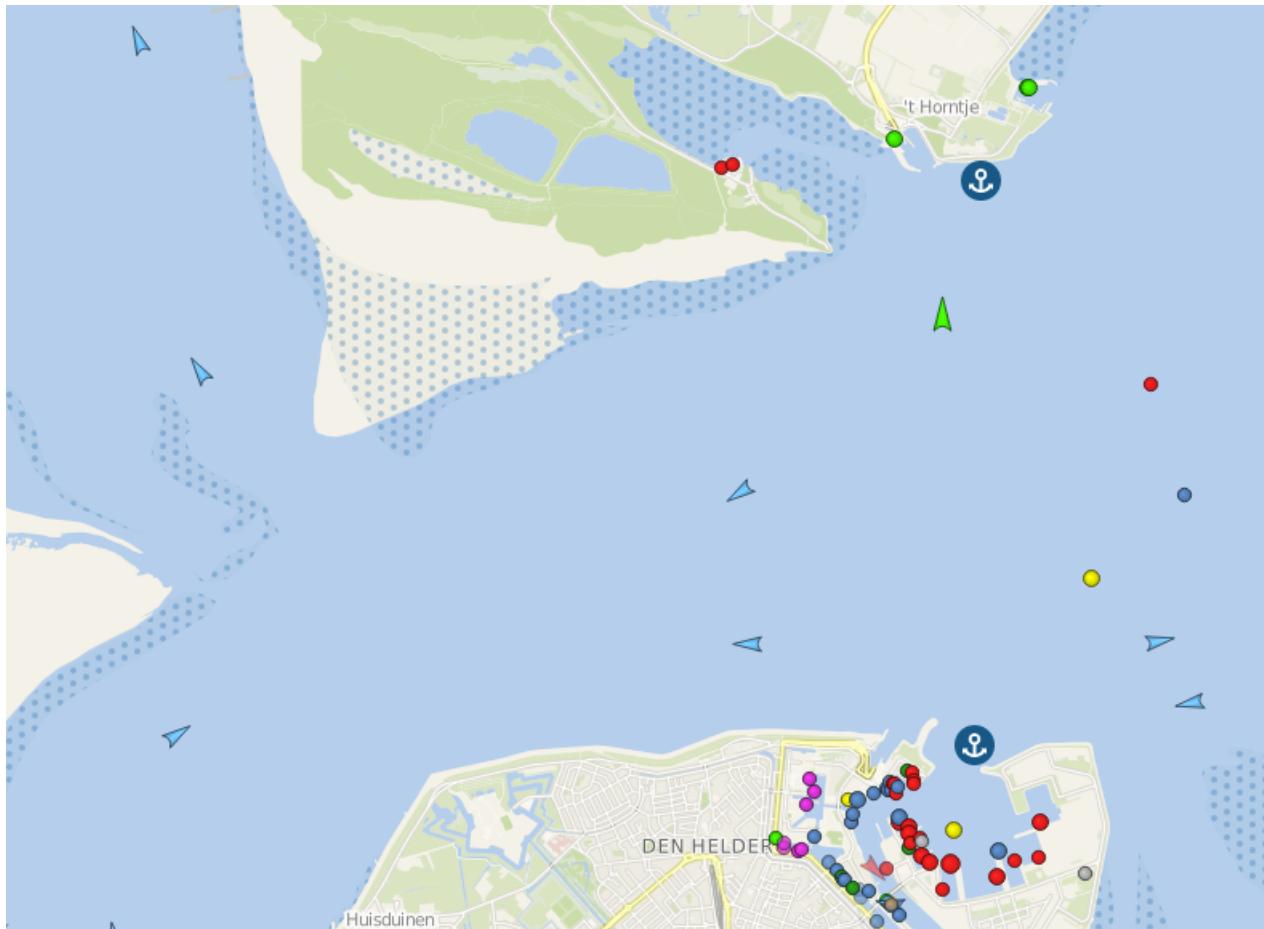


Top Software Technology

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



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

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
```

- data types matching the grammar

Gen

```
= Ship Name Gen
| WindPark Name Gen
| Cond Cond Gen
| Ret Expr
```

Cond

```
= LE Expr Expr
| And Cond Cond
| HasFlag Expr [Country]
| HasName Expr [Name]
```

Expr

```
= Var Name
| Num Real
| Distance Expr Expr
| Name Expr
```

web-editor in iTasks

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

derive class iTasks Gen, Cond, Res

genTask :: Task Gen

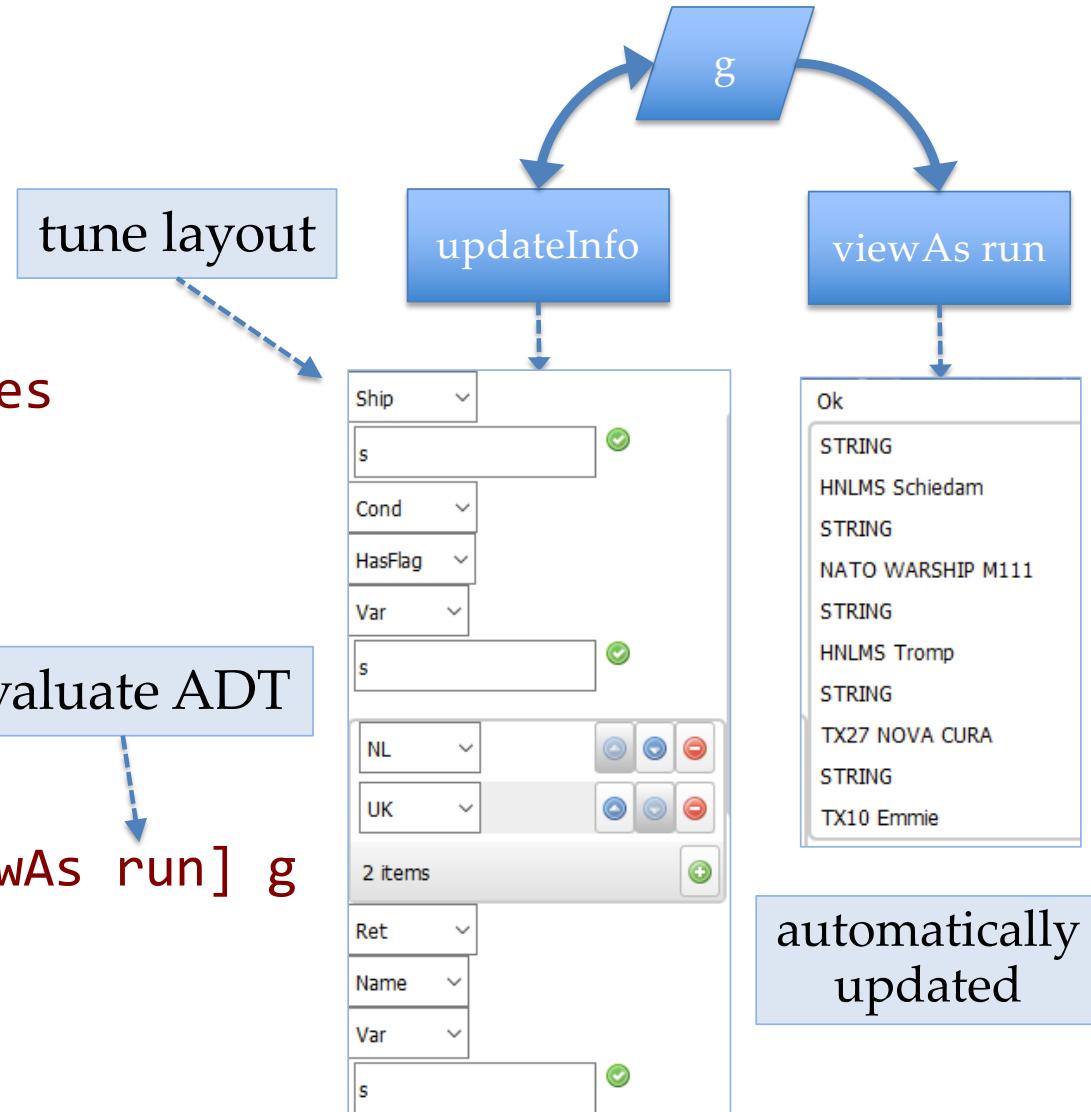
genTask =

 withShared e0 \g.

 updateSharedInformation [] g

 - || viewSharedInformation [ViewAs run] g

Start w = doTasks genTask w



how this works

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

user defined types

RIGHT (CONS (LEFT (RIGHT CONS ...



datatype-generic version

```
<div class="itasks-container var-cons" style="flex">
  <select class="itasks-component itasks-wrap-width itasks-wrap-height" style="flex">...</select> event
<div class="itasks-container var-cons" style="flex">
  <select class="itasks-component itasks-wrap-width itasks-wrap-height" style="flex">...</select> event
  <div class="itasks-container var-cons" style="flex">...</div> flex
</div>...
```



task-oriented programming framework

HTML + WA

A screenshot of a web-based graphical user interface for editing code. It shows a vertical stack of dropdown menus labeled 'Ret', 'Name', and 'Var', each with a downward arrow. Below these is a text input field containing the letter 's'. To the right of the input field is a green circular button with a checkmark. The entire interface is styled with a light blue border and white background.

GUI: web-editor



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

cannot be prevented by design of
datatypes if we need overloading

HasFlag (Num 42.0) [NL]

Distance (Num 7.0) (String "Hello World!")

LE (Var "a") (Num 137.0)

Var "undefined name"



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
-
- The diagram consists of a vertical brace on the left side, grouping the five numbered options. Each option is connected by a dashed blue arrow pointing to a rectangular box on the right. The boxes are arranged vertically from top to bottom. The top box is light blue and contains the text "too late: prevent errors". The middle box is light blue and contains the text "too complex for datatype generic programming". The third box from the top is red and contains the text "iTasks cannot derive this". The bottom two boxes are light blue and contain the text "implemented here".
- 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

```
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 =
```

```
[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
 ]
```

transforms any type to Dynamic

use editor recursively for arguments

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

for a Bool result



dynamic editors

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

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

```
:: VarName a = VarName String
```

2. select variables from a pool of defined variables

```
:: Bind = {idnt::String, val::Dynamic}
```

```
state0 :: [Bind]
```

```
state0 =[ {idnt = "s", val = dynamic ship0}
          , {idnt = "w", val = dynamic windpark0}
          ..
```

Identifiers	
Idnt	Val
s	Ship
t	Ship
w	WindPark
v	WindPark
p	Port

Add new identifier

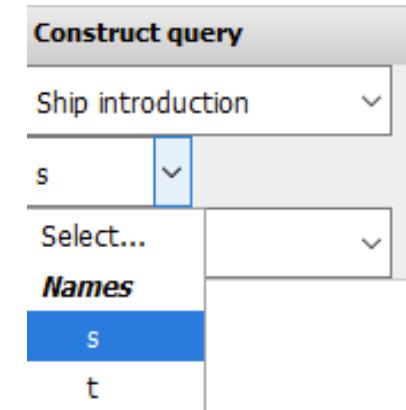
Idnt:
myShip

Val:
Ship



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} ← match dynamic of type t  
  = de idnt (dynamic (VarName idnt) :: VarName t)
```



obtained query DSL for ships

The screenshot shows a user interface for constructing a query. On the left, the "Construct query" panel contains dropdown menus for "Ship introduction" (selected: "Ship introduction"), "Condition" (selected: "Has flag"), and "Return name" (selected: "Name ship"). Below these are dropdowns for "s" and "s". In the center, the "Result of current query" panel displays a list of results under the heading "Ok":

Type	Value
STRING	Texelstroom
STRING	HNLMS Schiedam
STRING	NATO WARSHIP M111
STRING	HNLMS Tromp
STRING	TX27 NOVA CURA
STRING	TX10 Emmie

On the right, the "Identifiers" panel lists identifiers:

Idnt	Val
s	Ship
t	Ship
w	WindPark
v	WindPark
p	Port

Below this is an "Add new identifier" section with fields for "Idnt:" (myShip) and "Val:" (Ship), and a "Add" button.

updated if
query changes

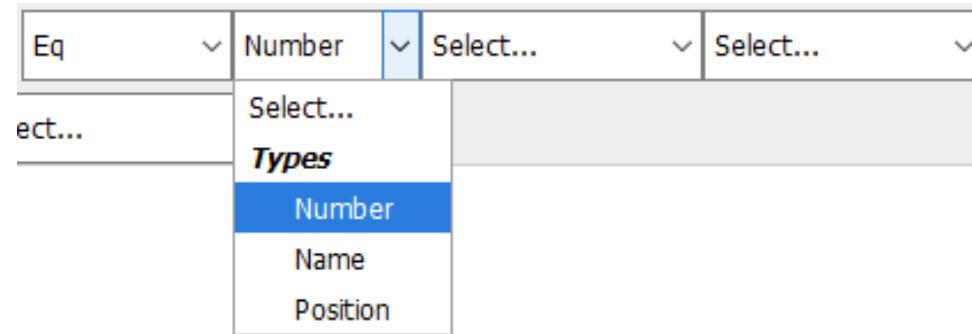


operator overloading

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

```
[ de "Eq"
  (dynamic \((EQ f) x y = f x >>= \a. y >>= \b.pure (a == b)
   :: A.a: (EQ a) (Eval a) (Eval a) -> Eval Bool)
 , de "Number" (dynamic EQ id :: EQ Real) ...
:: EQ a = EQ ((Eval a)->Eval a) & == a
```

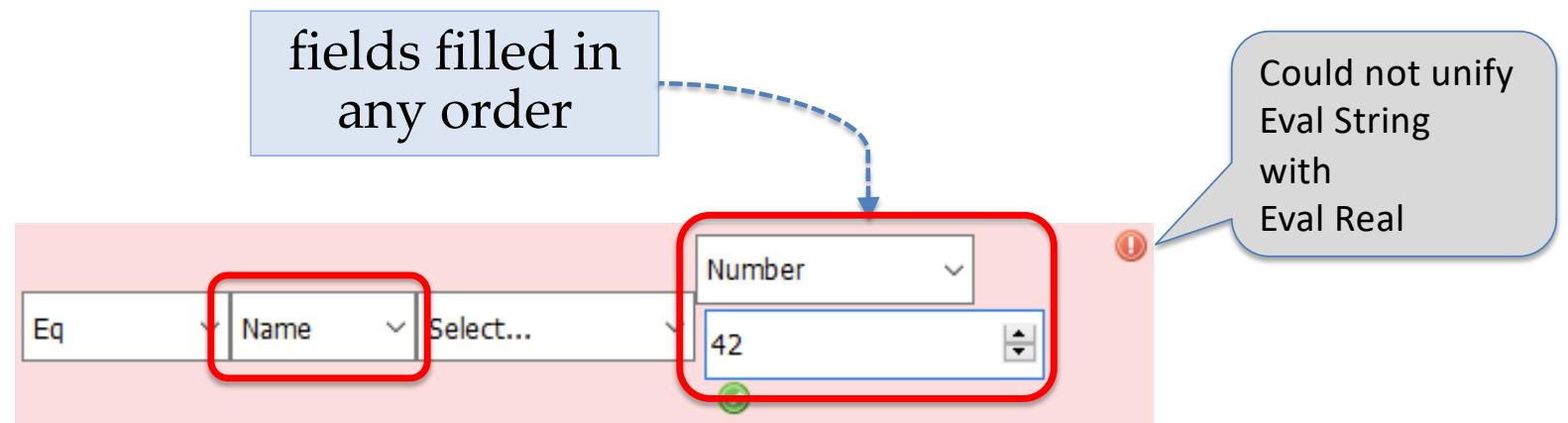
equal result types is context sensitive !



overloading errors

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

```
(dynamic \((EQ f) x y = f x >>= \a. y >>= \b.pure (a == b)
:: A.a: (EQ a) (Eval a) (Eval a) -> Eval Bool)
```



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`

DEMO ??

???



future work

- define variables on-the-fly:
the new changing state in the editors allows this
- more flexible editors:
change an expression x in Add $\ x \dots$
- composable editors
instead of one list of items for every editor
- ...

