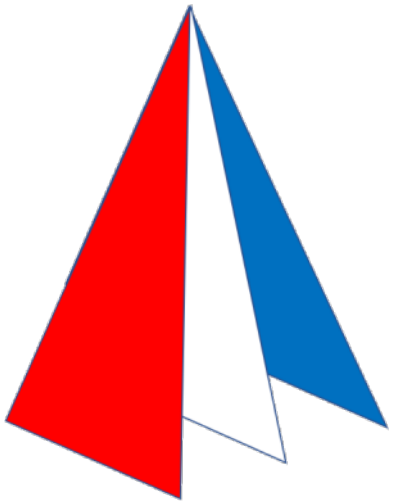
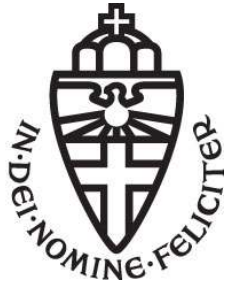
interactive creation of well-typed expressions in Domain Specific Languages

Pieter Koopman[1], Steffen Michels[2], Rinus Plasmeijer[1,2]

1: Radboud University Nijmegen
2: top-software.nl



Radboud University



Top Software Technology

interactive creation of well-typed DSL expressions

- a Domain Specific Language, DSL, is programming language specialized to a particular domain
  - e.g. queries over ships and their risks for the coast guard
  - Task Oriented Programming, TOP, for workflows
  - co-operative devices for the Internet of Things, IoT
  - tax forms and associated calculations
  - …
- our DSLs are embedded inside another language: eDSL
  - inherit all operations of the host language
  - we are particularly focused on **strong typing**
- focus of today: interactive creation of DSL expressions
  - e.g. dynamic creation of a query over ships
  - can we avoid parsing, type-checking and dynamic linking?

editor for deep embedded DSL

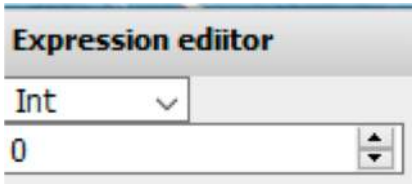
- running example with integers and Booleans

like the context free grammar

```
:: Expr = Int Int       | Bool Bool       | Add Expr Expr
        | And Expr Expr | Eq   Expr Expr | If  Expr Expr Expr
```

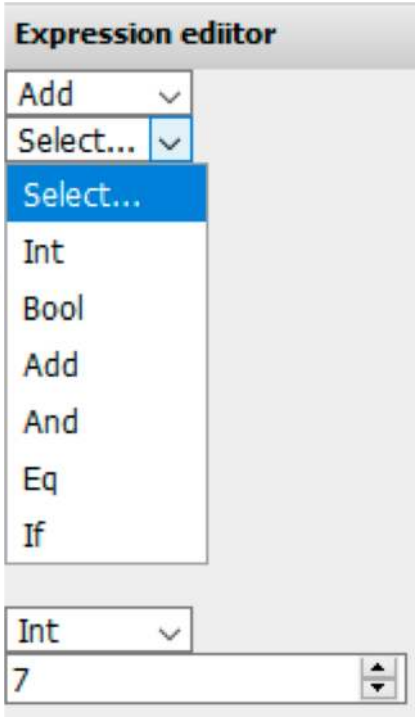
- making a browser-based editor for this type:

```
derive class iTask Expr
Start world = doTasks (updateInformation [] (Int 0)) world
```



a web-editor for Expr!
What is our problem ?

correct strong typing !



iTask editor magic: generic programming

type driven generic functions:

```
:: PAIR a b = PAIR a b
:: EITHER a b = LEFT a | RIGHT b
:: CONS a = CONS a
:: UNIT = UNIT
```

- 1 algorithm for all datatypes
- uniform encoding of types
- define operation (here `edit`) for `PAIR`, `EITHER`, `Int` etc.
- implicit transformation to and from generic type

```
:: Expr = Int Int       | Bool Bool      | Add Expr Expr
        | And Expr Expr | Eq   Expr Expr | If  Expr Expr Expr
:: GExpr = EITHER (CONS INT)                  // for Int Int
          (EITHER (CONS BOOL)                 // for Bool Bool
          (EITHER (CONS (PAIR Expr Expr))     // for Add Expr Expr
          ...
```

- Clean derives editor for `Expr` based on generic editors

limitations of this approach

```
:: Expr = Int Int       | Bool Bool       | Add Expr Expr
        | And Expr Expr | Eq   Expr Expr | If  Expr Expr Expr
```

- allows expression that are well-type in Clean, but incorrectly typed in the `Expr`-DSL
- some ill-typed DSL expressions

```
Add (Int 42) (Bool True)           // Add :: Int Int -> Int
And (Int 42) (Bool True)           // And :: Bool Bool -> Bool
If  (Int 42) (Bool True) (Int 7)   // If :: Bool a a -> a
Eq  (Int 42) (Bool True)           // Eq ::  a a -> Bool
```

required type

we need overloading

- we want a static type system spotting these errors
- **we do not want to write our own type-checker !**

dynamics to the rescue

- any type can be packed into a `Dynamic`

```
list :: [Dynamic]
list = [dynamic 7, dynamic (fib,4) :: (Int->Int,Int)
       ,dynamic (s2i,"30"), dynamic "1"]
```

we can add a type if we want

- dynamic type match to unpack dynamics type-safely

```
dSum :: [Dynamic] -> Int
dSum [x::Int:          rest] = x + dSum rest
dSum [t::(a->Int, a):rest] = (fst t) (snd t) + dSum rest
dSum [x:               rest] = dSum rest
dSum [] = 0
```

we can add cases by need

- application `Start = dSum list` 42

- **these dynamics can implement our runtime type-checker!**

dynamic editors

- plan:

1. programmer specifies list of labelled dynamic functions
   - each dynamic specifies a typed DSL construct
   - additional details for grouping, layout etc.
2. system selects items that produce required result type
3. user selects option in GUI based on label
4. dynamic editor used recursively for function arguments

- options to avoid DSL type-errors

1. better type information in editor
2. better typed DSL

- this is more work than generic deriving an editor,
  but ensures that we obtain correctly typed DSL expressions

1: better type information in editor

```
:: Expr = Int Int          | Bool Bool       | Add Expr Expr
        | And  Expr Expr | Eq   Expr Expr | If  Expr Expr Expr
:: Typed a b = Typed a    // b holds additional type information
```

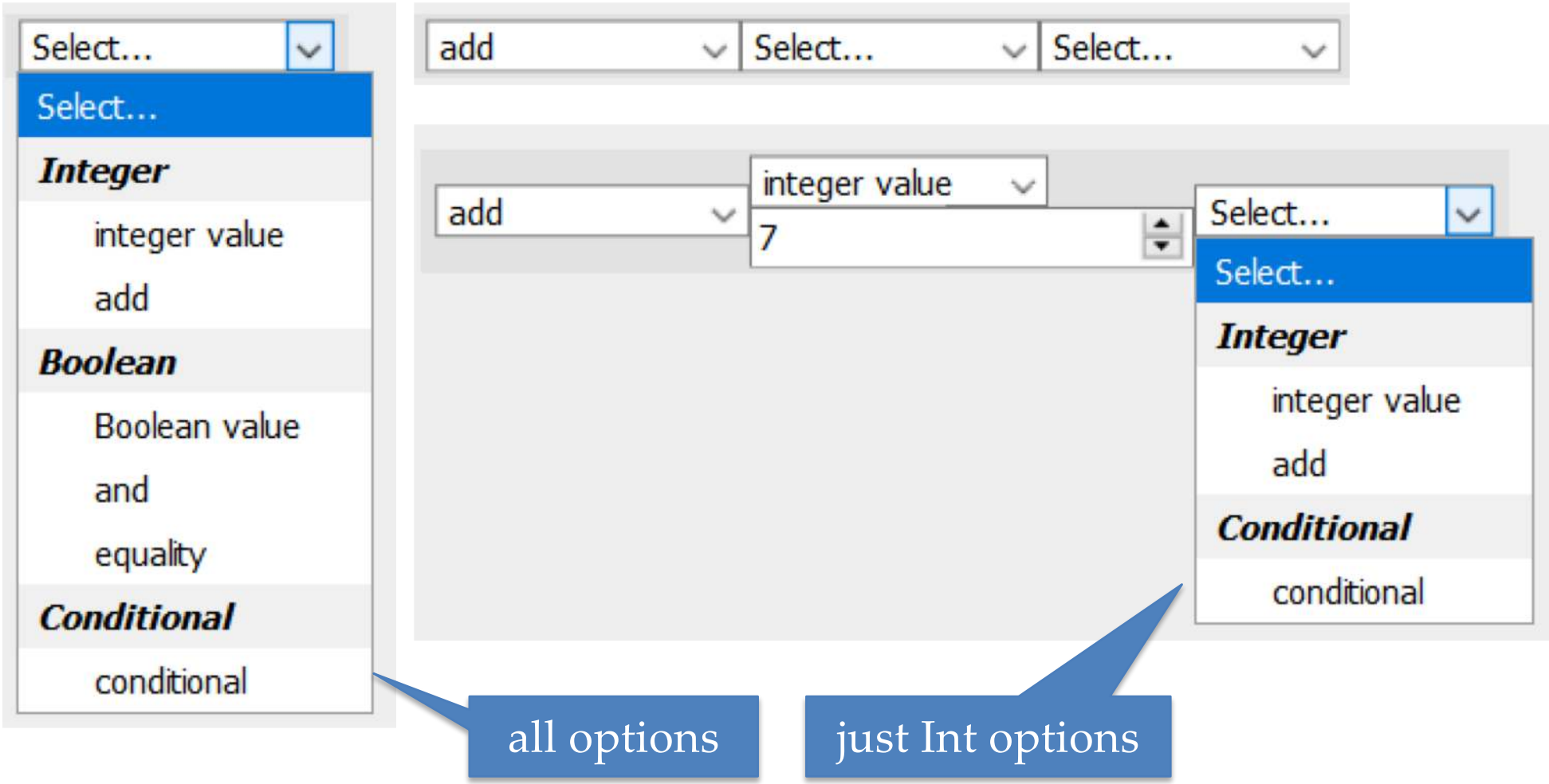
- dynamic editor cases needed

```
[de "int value" (dynamic \i.Typed (Int i) :: Int ->Typed Expr Int)
,de "Bool val" (dynamic \b.Typed (Bool b) :: Bool->Typed Expr Bool)
,de "add" (dynamic \(Typed x) (Typed y) -> Typed (Add x y) ::
 (Typed Expr Int) (Typed Expr Int) -> Typed Expr Int)
,de "equality" (dynamic \(Typed x) (Typed y).Typed (Eq x y) ::
 A.a: (Typed Expr a) (Typed Expr a) -> Typed Expr Bool)
,de "if" (dynamic \(Typed c) (Typed t) (Typed e).Typed (If c t e)::
 A.a:(Typed Expr Bool) (Typed Expr a) (Typed Expr a)->Typed Expr a)
..
```

no errors

this approach prevents all type problems

1: better type information in editor 2



- DSL is unchanged
- user can make only DSL-type-correct instances: **success !**

overloading in the DSL

- in our DSL `Eq` and `If` should be overloaded:

```
,de "equality" (dynamic \(Typed x) (Typed y).Typed (Eq x y) ::
 A.a: (Typed Expr a) (Typed Expr a) -> Typed Expr Bool)
,de "if" (dynamic \(Typed c) (Typed t) (Typed e).Typed (If c t e)::
 A.a:(Typed Expr Bool) (Typed Expr a) (Typed Expr a)->Typed Expr a)
```

- based on type `Typed Expr a` any `Expr` is allowed
  - the dynamic editor does allow any `Expr`
  - it uses dynamics to unify the arguments
  - indicates an error as soon as unification fails

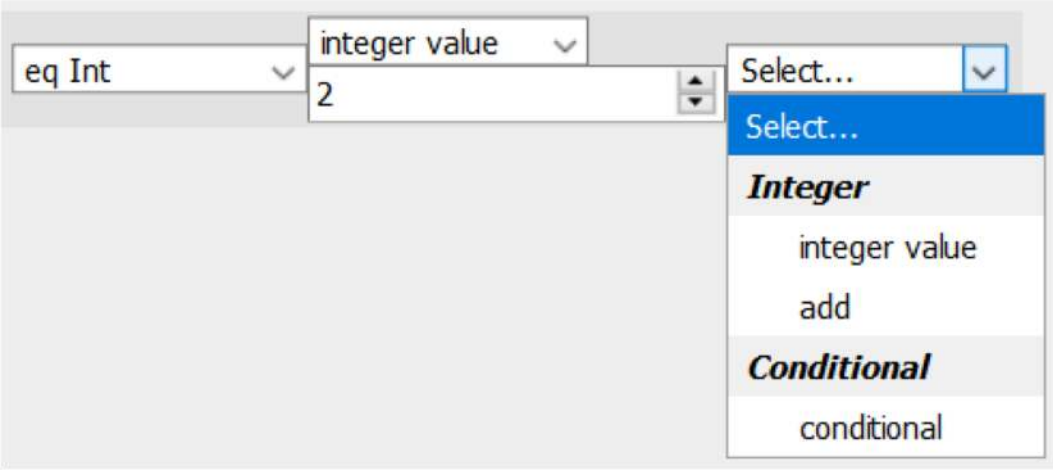
Could not unify
Typed Expr Int
with
Typed Expr Bool

preventing unification errors

- we can prevent unification by specialisation of type

```
,de "eq int" (dynamic \(Typed x) (Typed y).Typed (Eq x y) ::
 (Typed Expr Int) (Typed Expr Int) -> Typed Expr Bool)
,de "eq Bool" (dynamic \(Typed x) (Typed y).Typed (Eq x y) ::
 (Typed Expr Bool) (Typed Expr Bool) -> Typed Expr Bool)
```

- now the dynamic editor knows the type of arguments: no dynamic unification



- number of cases explodes if we have many types

2: better typed DSL – GADT based

```
:: Expr = Int Int          | Bool Bool       | Add Expr Expr
        | And  Expr Expr | Eq   Expr Expr | If  Expr Expr Expr
```

- replace this by:

```
:: Expr a
  = Lit a
  | Add (BM a Int)  (Expr Int)  (Expr Int)
  | And (BM a Bool) (Expr Bool) (Expr Bool)
  | E.b: Eq (BM a Bool) (Expr b) (Expr b) & == b
  | If  (Expr Bool) (Expr a) (Expr a)
```

- for type conversions (a poor man's GADT)

```
:: BM a b = {ab :: a -> b, ba :: b -> a}
```

- the only instance used

```
bm :: BM a a
bm = {ab = id, ba = id}
```

- example

```
Add bm (Lit 6) (Lit 36)
```

use a dynamic editor

generics cannot handle:
1. functions in record `BM`
2. quantified variables `E.b: ..`
3. class constraints `& == b`

2: better typed DSL – GADT based 2

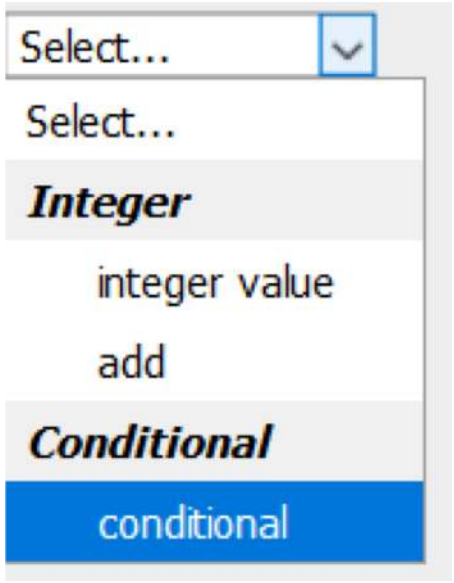
argument a indicates the type, we do not need `:: Typed a b = Typed a`

```
:: Expr a
  = Lit a
  | Add (BM a Int)  (Expr Int)  (Expr Int)
  | E.b: Eq (BM a Bool) (Expr b) (Expr b) & == b

[ de "integer value"
  (dynamic \i -> Lit i :: Int -> Expr Int)
, de "add"      (dynamic \x y -> Add bm x y ::
                 (Expr Int) (Expr Int) -> Expr Int)
, de "eq Int"   (dynamic \x y -> Eq bm x y ::
                 (Expr Int) (Expr Int) -> Expr Bool)
, de "eq Bool" (dynamic \x y->Eq bm x y ::
                 (Expr Bool) (Expr Bool) -> Expr Bool)
...
```

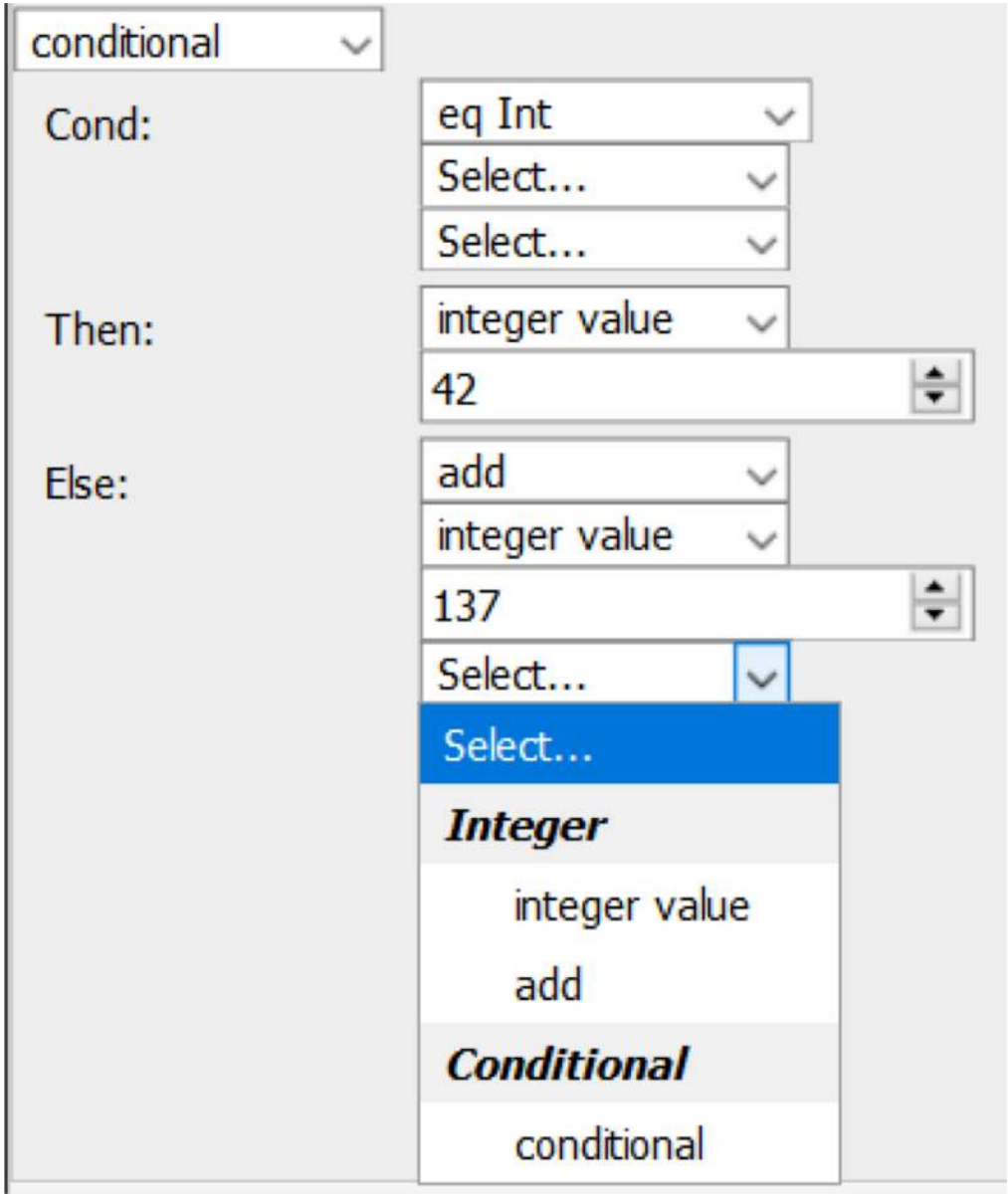
the GADT approach makes the editor simpler

2: better typed DSL 2



Correct DSL types are enforced by host language compiler

Not only during edit



DSL-variables

challenges:

- type of the variable
- existence of appropriate variable definition before use
  - even a GADT cannot guarantee this

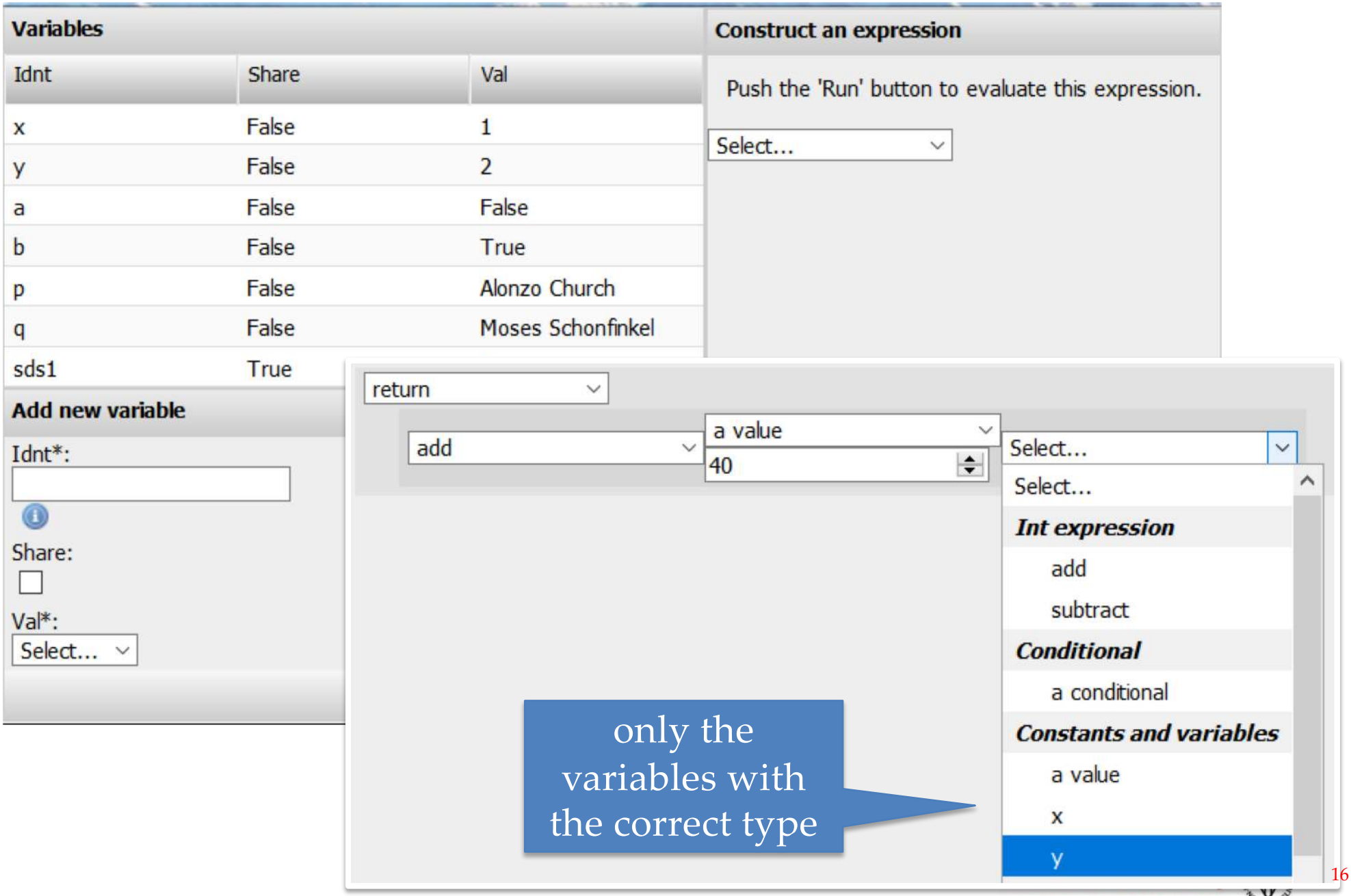
```
:: Expr a
   = Lit a
   | Var String
   | Add (BM a Int)  (Expr Int)  (Expr Int) | ...
```

our solution:

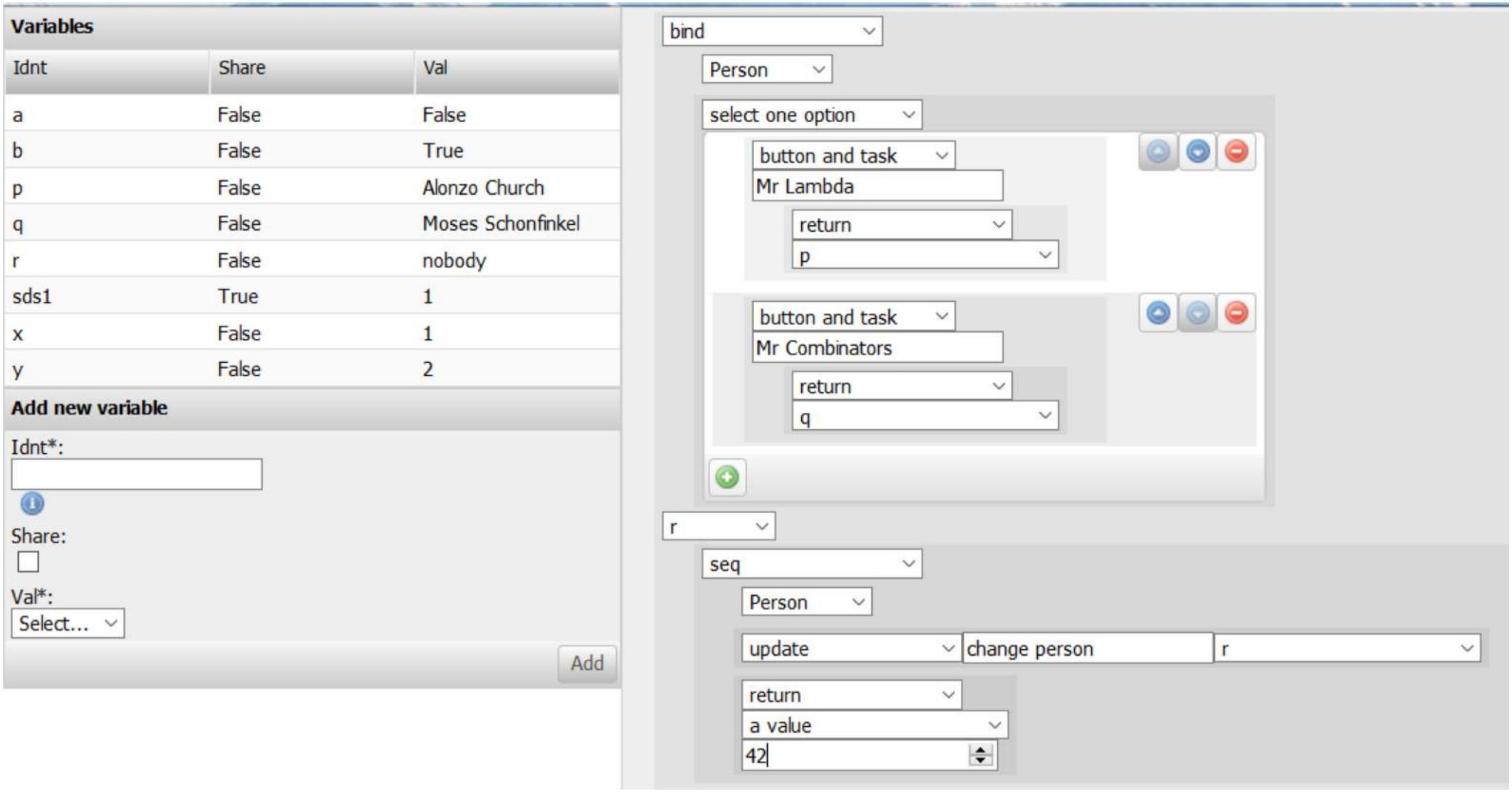
- define a shared pool of typed variables in the editor
- dynamic editor selects well-typed items from this pool
  - in iTasks we can edit the pool and the expression concurrently

better editors can make this pool on the fly

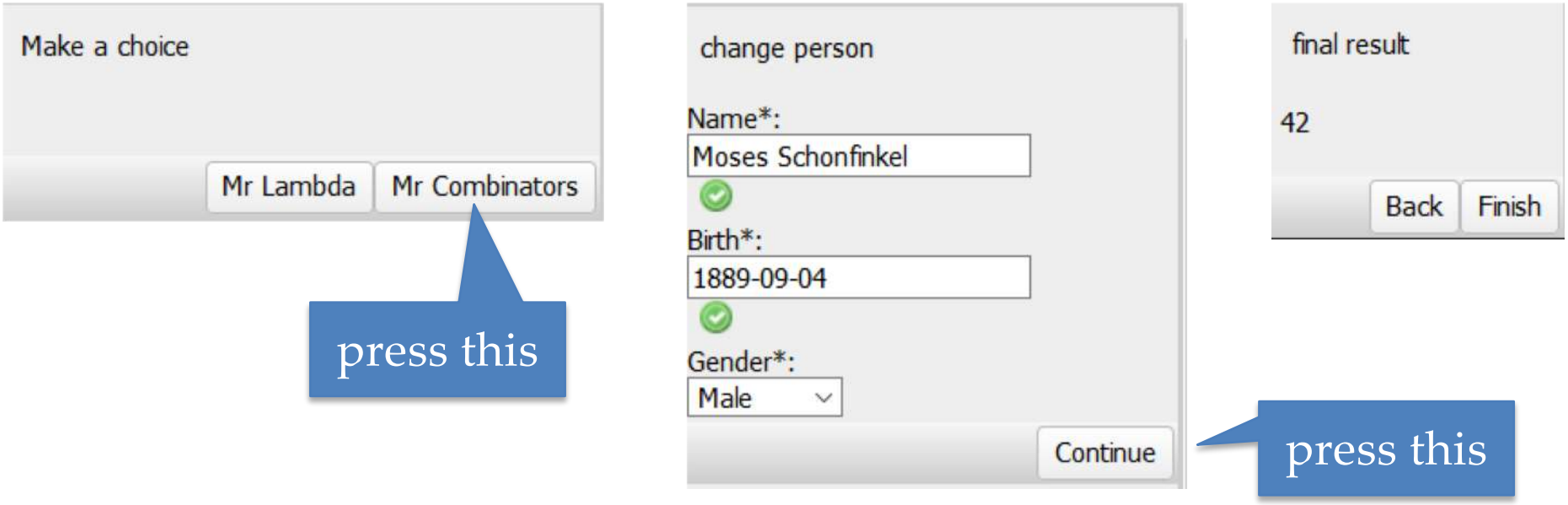
DSL-variables 2



more serious example: interactive workflow editor



more serious example: executing the composed task



- this show an restricted editor for iTask
- we can immediately execute the created task
- we do not want to make this a full programming language, but dynamic creation of limited tasks is very useful

ask a demo in the breaks

conclusion

- our goal: create well-typed DSL expressions dynamically
- iTask can derive editors for plain data-types,
  but this allows ill-typed DSL expressions
- the dynamic type-system can solve our type problems
- two approaches:

1. add additional information to editor, but not to type
2. improve the type holding the DSL, generics cannot handle it

- use a pool of typed variables to guarantee type and existence
- more tricks in the forthcoming paper

better editors can make this pool on the fly

ask a demo in the breaks

shallow embedded DSL

- shallow embedding is based on functions
- dynamic editors make datatypes,
  but we can use the results immediatly
- type for all results:

```
:: Val a = Val a
```

- relevant parts of the dynamic editor

```
[de "int val" (dynamic \i -> Val i :: Int -> Val Int)
,de "Boolean" (dynamic \b -> Val b :: Bool -> Val Bool)
,de "add"     (dynamic \(Val x) (Val y) -> Val (x + y) ::
                       (Val Int) (Val Int) -> Val Int)
,de "and"     (dynamic \(Val x) (Val y) -> Val (x && y) ::
                       (Val Bool) (Val Bool) -> Val Bool)
,de "eq int"  (dynamic \(Val x) (Val y) -> Val (x == y) ::
                       (Val Int) (Val Int) -> Val Bool)
,de "if"      (dynamic \(Val c) t e -> if c t e ::
                       A.a: (Val Bool) (Val a) (Val a) -> Val a)
```