Embedded Domain Specific Languages, part 3/4

Sven-Bodo Scholz, **Peter Achten**

Advanced Programming

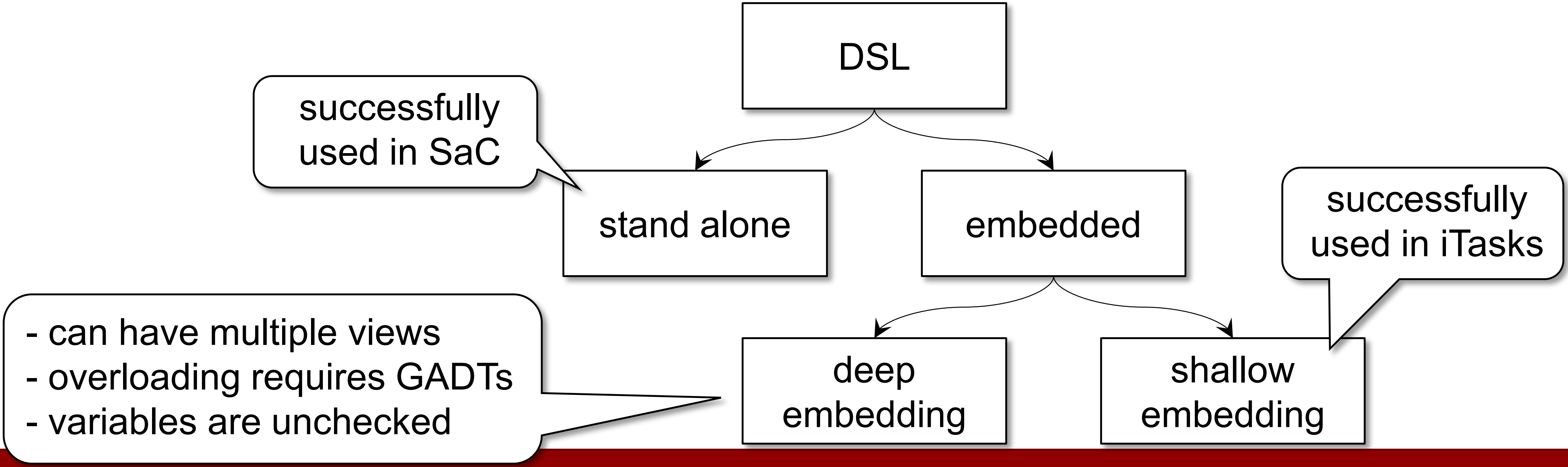
(based on slides by Pieter Koopman)

What this lecture is about

- Shallowly Embedded Domain Specific Languages

Domain-specific language (DSL)

- DSL is specialized to a particular application domain
  - e.g. HTML, TeX, iTask, SVG, SaC, ITTT, …
  - in contrast to a general-purpose language (GPL)
- Implementation strategies



Deep AExpressions:
DSL is a set of data structures + evaluator

```
:: AExpr
 = Int Int
 | Var Var
 | (+.) infixl 6 AExpr AExpr
 | (-.) infixl 6 AExpr AExpr
 | (*.) infixl 7 AExpr AExpr
:: Var :== String

A :: AExpr State -> Int

Start = A (Int 6 *. Var "x") sx
sx    = ("x" |-> 7) emptyState
```

enrich with GADTs, to increase type correctness

structure evaluator monadically

efficient state with dynamics and $\mathcal{O}({}^2\log(n))$ lookup (`Data.Map`)

Shallow AExpressions: DSL is a set of functions

```
:: SemA :== State -> Int

int :: Int -> SemA
int n = \s = n

var :: Var -> SemA
var v = \s = s v

instance + SemA where (+) x y = \s = x s + y s
instance - SemA where (-) x y = \s = x s - y s
instance * SemA where (*) x y = \s = x s * y s

Start = (int 6 * var "x") sx
sx    = ("x" |-> 7) emptyState
```

no plain `Int`,
we need a `SemA`

no `eval`,
just pass the state

here we can use the
ordinary `+`, `-` and `*`

Parameterized semantics to handle multiple types

```
:: Sem a :== State -> a

lit :: a -> Sem a
lit a = \s = a

var :: Var -> Sem Int
var v = \s = s v

instance + (Sem a) | + a where (+) x y = \s = x s + y s

(&&.) infixr 3 :: (Sem Bool) (Sem Bool) -> Sem Bool
(&&.) x y = \s = x s && y s

(==.) infix 4 :: (Sem a) (Sem a) -> Sem Bool | == a
(==.) x y = \s = x s == y s
```

to allow various types

includes `True` and `False`

we have overloading ☺

we need new operators ☹

Hiding the state in a monad and error handling

```
:: State s a =: S (s -> (MaybeError String a, s)) // from Data.Error

instance Monad (State s)
   where bind



instance Functor (State s)
   where fmap

instance pure (State s)
   where pure

instance <*> (State s)
   where (<*>)

instance MonadFail (State s)
   where fail
```

Hiding the state in a monad and error handling

```
:: State s a =: S (s -> (MaybeError String a, s)) // from Data.Error

instance Monad (State s)
   where bind (S f) g
            = S \s = case f s of
                       (Ok a,     t) = let (S h) = g a in h t
                       (Error e, t) = (Error e, t)
instance Functor (State s)
   where fmap f (S g) = S \s = let (a, t) = g s in (fmap f a, t)
instance pure (State s)
   where pure a = S \s = (Ok a, s)
instance <*> (State s)
   where (<*>) f x = f >>= \g = x >>= \a = pure (g a)
instance MonadFail (State s)
   where fail e = S \s = (Error e, s)
```

An efficient state that contains arbitrary types

```
:: State s a =: S (s -> (MaybeError String a, s)) // from Data.Error
:: Sem a :== State Env a
:: Env   :== 'Data.Map'.Map Int Dynamic
:: Var a :== Int

read :: (Var a) -> Sem a | TC a
read v
  = S \env = case 'Data.Map'.get v env of
                 ?Just (x::a^) = (Ok x, env)
                 ?Just x = (Error ("Var " <+ v <+ " of wrong type"), env)
                 ?None   = (Error ("Var " <+ v <+ " undefined"), env)

write :: (Var a) a -> Sem a | TC a
write v x = S \env = (Ok x,'Data.Map'.put v (dynamic x) env)

fresh :: Sem (Var a)
fresh = S \env = (Ok ('Data.Map'.mapSize env), env)
```

any object fits in a Dynamic

phantom type

Hiding the state in a monad, part 2

```
var :: (Var a) -> Sem a | TC a
var v = read v

lit :: a -> Sem a
lit x = pure x

instance + (Sem a) | + a where (+) x y = (+) <$> x <*> y

(&&.) infixr 3 :: (Sem Bool) (Sem Bool) -> Sem Bool
(&&.) x y = (&&) <$> x <*> y

(==.) infix 4 :: (Sem a) (Sem a) -> Sem Bool | == a
(==.) x y = (==) <$> x <*> y

// etc.
```

like before: there are semantic implications!

Statements

check on variable definition is still missing

```
skip :: Sem ()
skip = pure ()

(:=.) infixr 2 :: (Var a) (Sem a) -> Sem a | TC a
(:=.) v x = x >>= write v

(:.) infixr 1 :: (Sem a) (Sem b) -> Sem b
(:.) x y = x >>| y

If :: (Sem Bool) Then (Sem a) Else (Sem a) -> Sem a
If c Then t Else e = c >>= \b = if b t e

while :: (Sem Bool) (Sem a) -> Sem ()
while c b = c >>= \x = if x (b :. while c b) (pure ())
```

overloaded, allows variables of type `Bool`

this allows C-like assignments `x :=. y :=. lit 42`

Type-safe variable definition

- Idea: use function argument to represent typed variable

```
def 7 \x = write x (read x + lit 1)
```

```
x :: Var Int
```

- Somewhat nicer syntax

```
def \x = 7 In x :=. var x + lit 1
```

constructor

write

read

```
:: In a b = In infix 0 a b
```

an infix tuple

```
def :: ((Var a) -> In a (Sem b)) -> Sem b | TC, < a
def f = fresh >>= \v = let (a In sem) = f v in write v a >>| sem
```

Example

```
fac :: Int -> Sem Int
fac x
  = def \r = 1 In
    def \n = x In
    while (lit 1 <. var n) (
       r :=. var r * var n :.
       n :=. var n - lit 1
    ) :.
    var r

run :: (Sem a) -> MaybeError String a
run (S f) = fst (f 'Data.Map'.newMap)

Start = run (fac 5)
```

```
Ok 120
```

this \ is the only strange thing in the DSL syntax

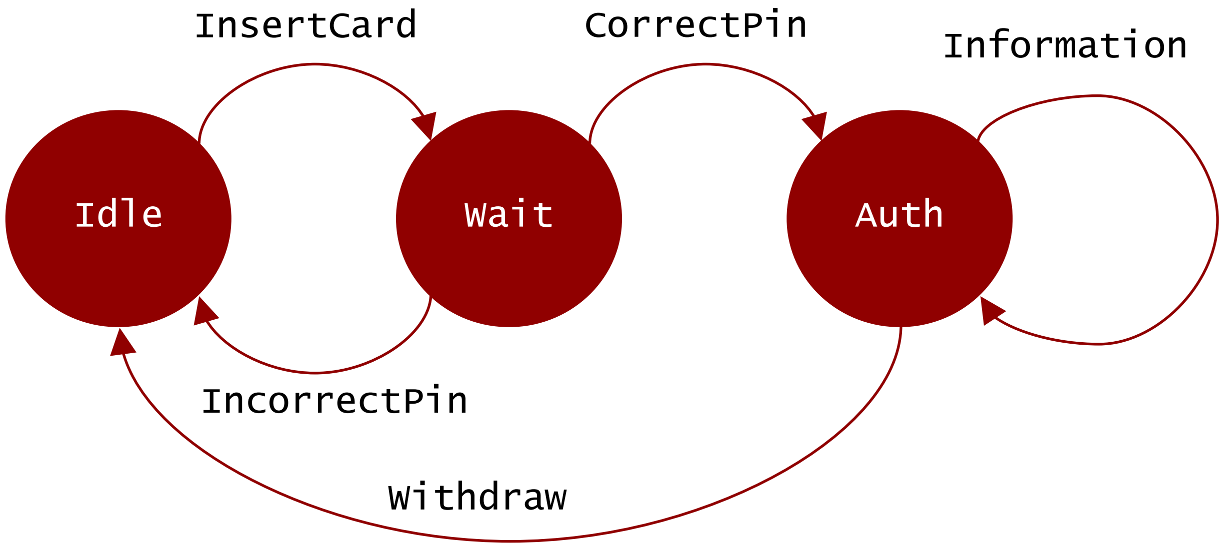
new DSL has:

- static typing
- overloading
- type-safe variables
- nice syntax

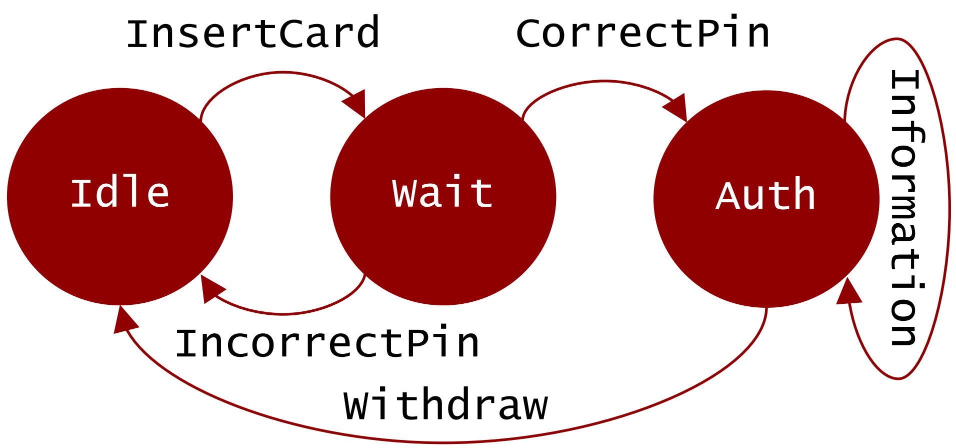
ATM

shallow embedded FSM

ATM




```
:: Idle = Idle
:: Wait = Wait
:: Auth = Auth
:: S a :== [String] // or any other state information
```



- Correctly type-checked:

```
p = insertCard :. correctPin :.
    withdraw 42
```

- Desired error:

```
q = insertCard :. withdraw 42
    cannot unify
    (S Wait) -> S v1
    (S Auth) -> S Idle
```

```
insertCard :: (S Idle) -> S Wait
insertCard l = ["insertCard" : l]

correctPin :: (S Wait) -> S Auth
correctPin l = ["correctPin" : l]

incorrectPin :: (S Wait) -> S Idle
incorrectPin l = ["incorrectPin" : l]

information :: (S Auth) -> S Auth
information l = ["information" : l]

withdraw :: Int (S Auth) -> S Idle
withdraw i l = ["withDraw " <+ i : l]

(:.) infixl 1 :: ((S a) -> S b)
                 ((S b) -> S c)
              -> ((S a) -> S c)
(:.) f g = g o f

runATM f = reverse (f [])
```

Functions
definitions can be more general

shallow embedded functional DSL

A functional DSL

- Finally, we have a type safe imperative DSL with type safe definitions

```
def :: ((Var a) -> In a (Sem b)) -> Sem b | TC, < a
def f = fresh >>= \v = let (a In sem) = f v in write v a >>| sem
```

- More general definitions

```
def :: (a -> In a (Val b)) -> Val b
def f = let (body In exp) = f body in exp
```

we reuse the expressions from above, no state!

the tuple is not just nice syntax, need for recursion

- This allows

```
e1 = def \x = lit 7 In x + lit 1              // x has type  Val Int
e2 = def \f = (\x = x + lit 1) In f (lit 0)   // f has type (Val Int) -> Val Int
e3 = def \f = (\n = If (n ==. lit 0)
                   Then (lit 1)
                   Else (n * f (n - lit 1))) In f (lit 4)
```

Expressions

this shows why it is useful to stack the `Monad` class

`-`, `*`, `/` similar

can also be more general

```
:: Val a :== MaybeError String a

lit :: a -> Val a
lit x = pure x

instance + (m a) | + a & Monad m where (+) x y = (+) <$> x <*> y

If :: (Val Bool) (Val a) (Val a) -> Val a
If c t e = c >>= \b = if b t e

(&&.) infixr 3 :: (Val Bool) (Val Bool) -> Val Bool
(&&.) x y = (&&) <$> x <*> y

(==.) infix 4 :: (Val a) (Val a) -> Val Bool | == a
(==.) x y = (==) <$> x <*> y

(<.) infix 4 :: (Val a) (Val a) -> Val Bool | < a
(<.) x y = (<) <$> x <*> y
```

Multiple arguments and nested functions

```
e4 = def \d = (\x y = y / x) In d (lit 0) (lit 1)
e5 = // compute xⁿ in O(log n) steps
     def \odd = (\n = If (n ==. lit 1) (lit True)
                   (If (n <. lit 1) (lit False) (odd (n - lit 2)))) In
     def \pow = (\x n = If (n <. lit 0)
                           (fail "pow x n: n must be >= 0")
                           (If (n ==. lit 0)
                               (lit 1)
                               (If (odd n)
                                   (x * pow x (n - lit 1))
                                   (def \y = pow x (n / lit 2) In y * y)))) In
     pow (lit 3) (lit 5)
```

these are type checked by the Clean compiler ☺

Multiple views

shallow embedding

all wonderful, but we have only a single view (evaluation)

Why are different views important?

```
def \r = 1 In
def \n = 3 In
while (lit 1 <. var n) (
     r :=. var r * var n :.
     n :=. var n - lit 1
 ) :.
var r
```

different **views**

variables
`r n`

result
6

optimized
`lit 6`

C code
```
int n = 3;
...
```

cost
3 multiplications

termination
yes

type
`Int`

Multiple views

```
:: Views a = {a :: Sem a, show :: [String]}
:: Sem   a :== State -> (?a, State)

show :: String (Views a) -> Views a
show str views = {views & show = [str : views.show]}

var :: Var -> Views Int
var v = {a = \s = (?Just (s v), s), show = [v]}

lit :: a -> Views a | toString a
lit a = {a = \s = (?Just a, s), show = [toString a]}
```

show is independent of evaluation

Monadic stuff

```
instance Functor Views
   where fmap f views
           = {views & a = \s = let (r, s2) = views.a s in (fmap f r, s2)}
instance pure Views
   where pure a = {a = \s = (?Just a, s), show = []}
instance <*> Views
   where <*> vf va = {a = \s = case vf.a s of
                                 (?Just f, s) = case va.a s of
                                    (?Just a, s2) = (Just (f a), s2)
                                    _ = (?None, s)
                                 _ = (?None, s)
                     ,show = vf.show ++ va.show
                     }

instance Monad Views
   where bind va f = …
```

forgot to add `toString` a?

yes, a continuation is more efficient

Multiple views

```
instance + (Views a) | + a where + x y = (+) <$> x <*> show "+" y

(&&.) infixr 3 :: (Views Bool) (Views Bool) -> Views Bool
(&&.) x y = (&&) <$> x <*> show "&&" y

(==.) infix 4 :: (Views a) (Views a) -> Views Bool | == a
(==.) x y = (==) <$> x <*> show "==" y

If :: (Views Bool) Then (Views a) Else (Views a) -> Views a
If c Then t Else e
  = { a = \s = case c.a s of
                 (?Just b, s1) = if b t.a e.a s1
                 (_, s1)       = (?None, s1)
    , show = ["if (" : c.show] ++ [") then {" : indent ["\n" : t.show]] ++
             ["} else {" : indent ["\n" : e.show]] ++ ["}"]
    }
```

evaluation and show separated

Shallow embedding multiple views

+

- it can be done
- safe variables
- overloading works
- more type control
- easy to add a language construct

-

- all views are always computed, even if we need only one
- no separation of concerns
- adding a view is a troublesome

Reification

multiple views

- turn functions into data structures
- shallow embedding produces a deep embedding

Reification

there are many different notions of reification

“Process that makes a computable / addressable object out of a non-computable / addressable one.” [wiktionary.org]

Shallow embedding
- functions
- structure hidden

Deep embedding
- data types
- structure visible

we know this

view 1
- evaluation

view 2
- printing

view 3
- optimization

Target is the GADT version of WHILE (1/2)

```
:: Expr a
 = Lit  a
 | Var    (BM a Int)  Var
 | Plus   (BM a Int)  (Expr Int) (Expr Int)
 | Sub    (BM a Int)  (Expr Int) (Expr Int)
 | Mul    (BM a Int)  (Expr Int) (Expr Int)
 | Not    (BM a Bool) (Expr Bool)
 | And    (BM a Bool) (Expr Bool) (Expr Bool)
 | E.b:Eq (BM a Bool) (Expr b)   (Expr b) & ==, toString b
 | Le     (BM a Bool) (Expr Int) (Expr Int)

:: Var :== Int

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

only `Int` variables, this is a choice, not a constraint

earlier we used `String`, `Int` is convenient to generate 'fresh' variables

does it matter if we use an ADT or a GADT?

Target is the GADT version of WHILE (2/2)

```
:: Stmt
 = AssignStmt Var  (Expr Int)
 | SeqStmt    Stmt Stmt
 | IfStmt     (Expr Bool) Stmt Stmt
 | WhileStmt  (Expr Bool) Stmt
 | SkipStmt
```

changed constructor names to use them in shallow embedding

Reification of expressions

we have seen this

```
var     = Var bm
true    = Lit True
false = Lit False
num n = Lit n

instance +   (Expr Int)  where (+) x y = Plus bm x y
instance *   (Expr Int)  where (*) x y = Mul  bm x y
instance -   (Expr Int)  where (-) x y = Sub  bm x y
instance ~   (Expr Bool) where ~   x   = Not  bm x
instance one (Expr Int)  where one     = num 1

(==.) infix 4 :: (Expr a) (Expr a) -> Expr Bool | ==, toString a
(==.) x y = Eq bm x y

(<.) infix 4 :: (Expr Int) (Expr Int) -> Expr Bool
(<.) x y = Le bm x y
```

Monadic tooling

we have seen this

```
:: SA s a  =:  SA (s -> (a, s))
:: Sem a    :== SA State a
:: Reify a :== SA Int a

fresh :: Reify Var
fresh = SA \x = (x, x + 1)

write :: Var Int -> Sem Int
write v x = SA \s = (x, (v |-> x) s)

instance pure     (SA s) where pure a = SA \s = (a,s)
instance Monad    (SA s) where bind (SA a) f = SA \i = let (b,j) = a i
                                                             (SA g) = f b
                                                          in g j
instance Functor (SA s) where fmap  f  sa = sa >>= \a = pure (f a)
instance <*>      (SA s) where (<*>) sf sa = sf >>= \f = fmap f sa
```

The shallow embedded WHILE DSL

```
:: In a b
 = In infix 0 a b
```

```
(:=.) infix 2 :: Var (Expr Int) -> Reify Stmt
(:=.) v e = pure (AssignStmt v e)

(:.) infixr 1 :: (Reify Stmt) (Reify Stmt) -> Reify Stmt
(:.) s t = SeqStmt <$> s <*> t

IF :: (Expr Bool) (Reify Stmt) (Reify Stmt) -> Reify Stmt
IF b t e = IfStmt b <$> t <*> e

While :: (Expr Bool) (Reify Stmt) -> Reify Stmt
While b s = WhileStmt b <$> s

Skip :: Reify Stmt
Skip = pure SkipStmt

def :: (Var -> In (Expr Int) (Reify Stmt)) -> Reify Stmt
def f = fresh >>= \v = let (a In s) = f v in v :=. a :. s
```

only `Int` variables

do not forget this

Example in reified shallow WHILE

```
facWhile n
 = def \x = num n In
   def \y = num 1 In
   While (~ (num 1 ==. var x))
   ( y :=. var y * var x :.
     x :=. var x - one
   )

run :: (Reify Stmt) -> State
run rs = snd (f emptyState)
where
    (SA f)    = evalStmt (fst (stmt 0))
    (SA stmt) = rs

Start = unSA (facWhile 4) 0
```

Reified DSL has:

- static typing
- overloading
- type-safe variables
- nice syntax
- **multiple views from GADT**

Discussion

- Shallow embedding: DSL is represented by functions

+ simple (kind of)
+ Clean compiler checks types in DSL (easier than GADT)
+ **variables are type-safe** (typed and defined)
+ overloading in the DSL is possible
    - equality, comparison, …
+ less repeated work
    - variables for `Int` and `Bool`
+ efficient (no interpretation overhead)

- hard to have many views (interpretations)
    - evaluation, printings, finding variables, optimization, simulation, …

- from datatype to shallow embedded DSL is next to impossible

- reification saves the day, but adds overhead and feels like cheating