Embedded Domain Specific Languages, part 1/4

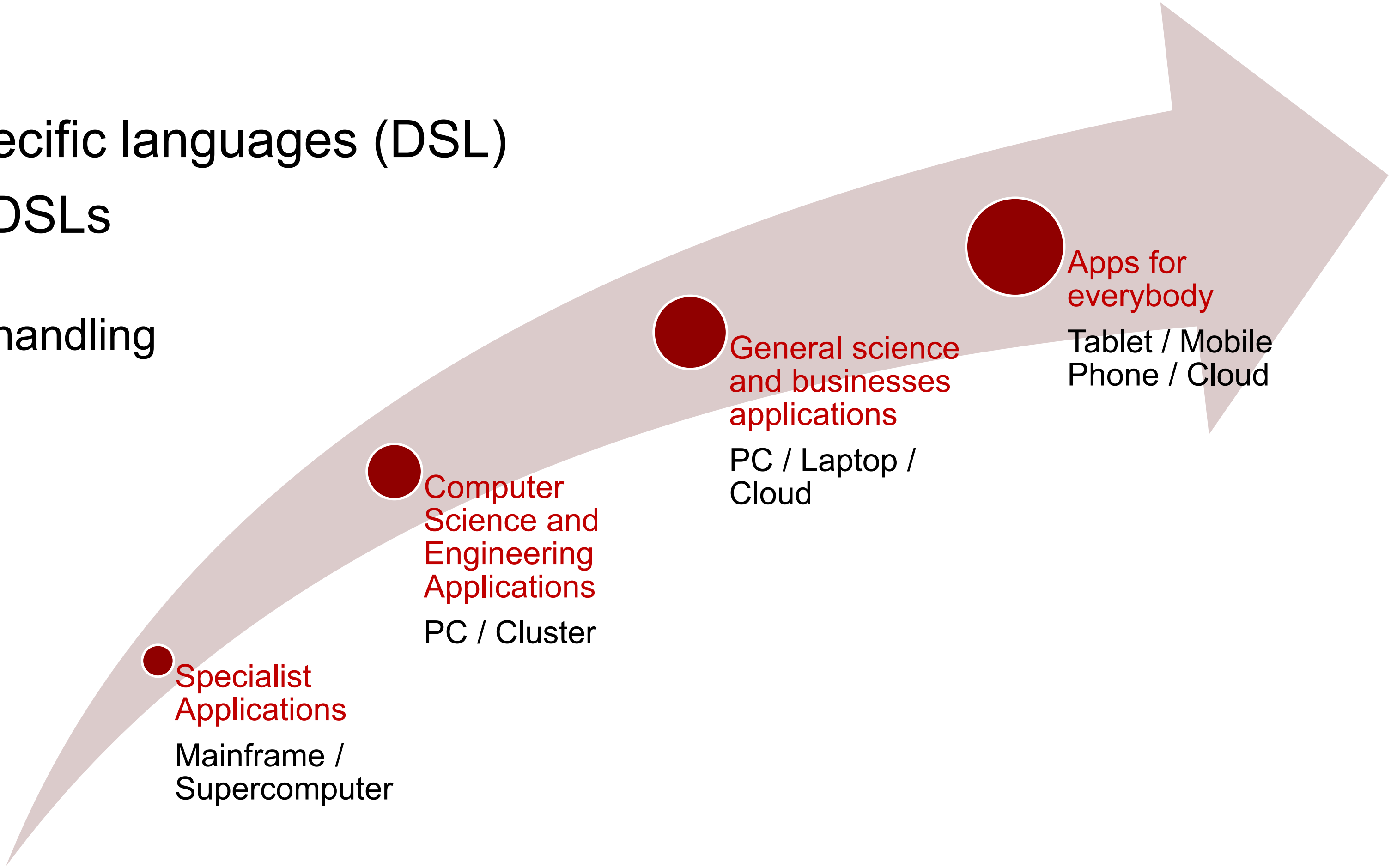
Sven-Bodo Scholz, **Peter Achten**

Advanced Programming

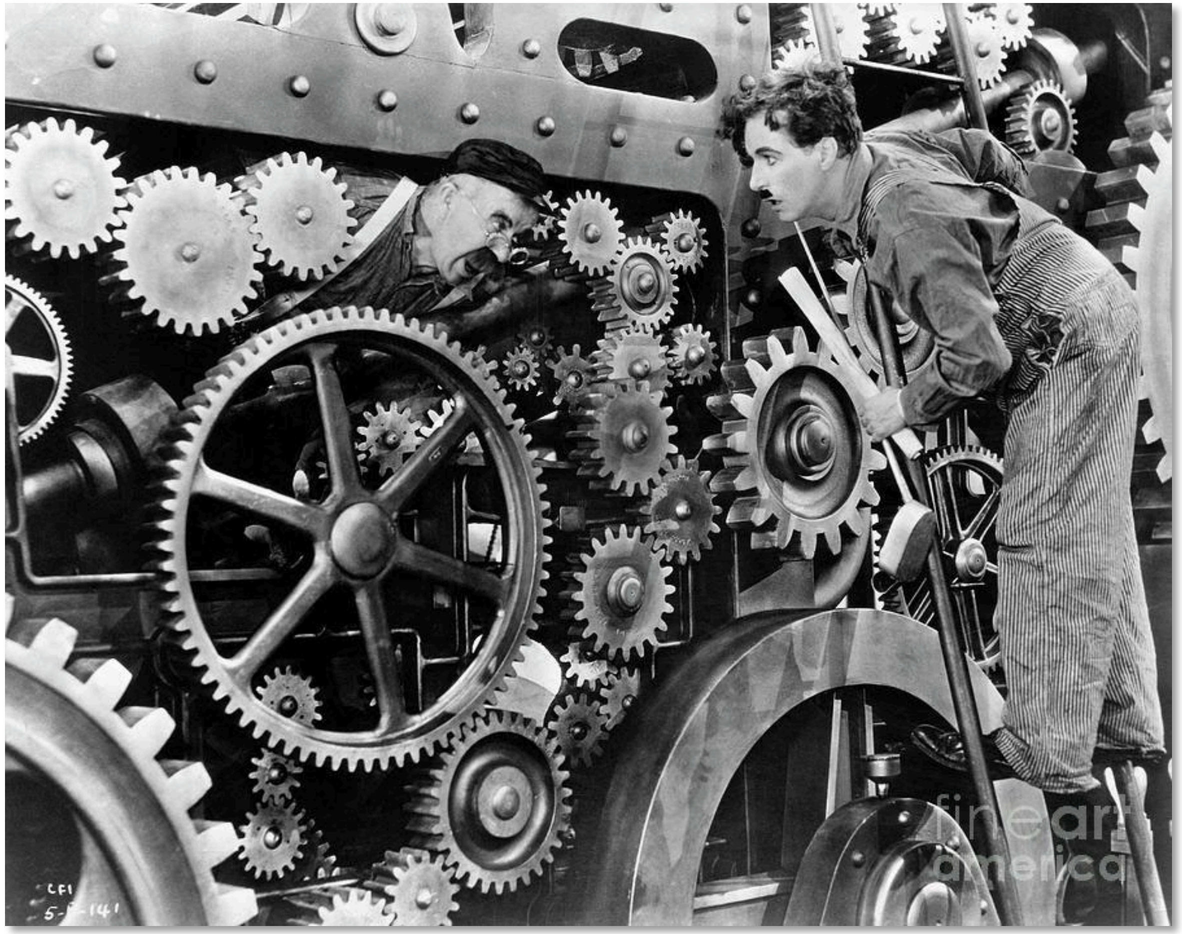
(based on slides by Pieter Koopman)

What this lecture is about

- Actually, the next four lectures
- High Productivity via domain specific languages (DSL)
- High Productivity of embedded DSLs
  - lecture 1/4: deep embedding
  - lecture 2/4: state abstraction and handling
  - lecture 3/4: shallow embedding
  - lecture 4/4: multi-view embedding



Productivity is key



- How long does it take me to write the first prototype?
- How long does it take me to write the full application?
- How much maintenance is needed?
- How adaptable is the code?
- How fast is the code?
- How robust is the code?

Using a DSL makes this all easier

We can use existing DSLs, or create tailor-made DSL for the problem of the day

DSLs are everywhere

- Array programming
- Task Oriented Programming
- Databases: SQL, …
- Web: HTML, web assembly, …
- LaTeX, TeX, …
- Tax office: input taxation, benefits (toeslagen), …

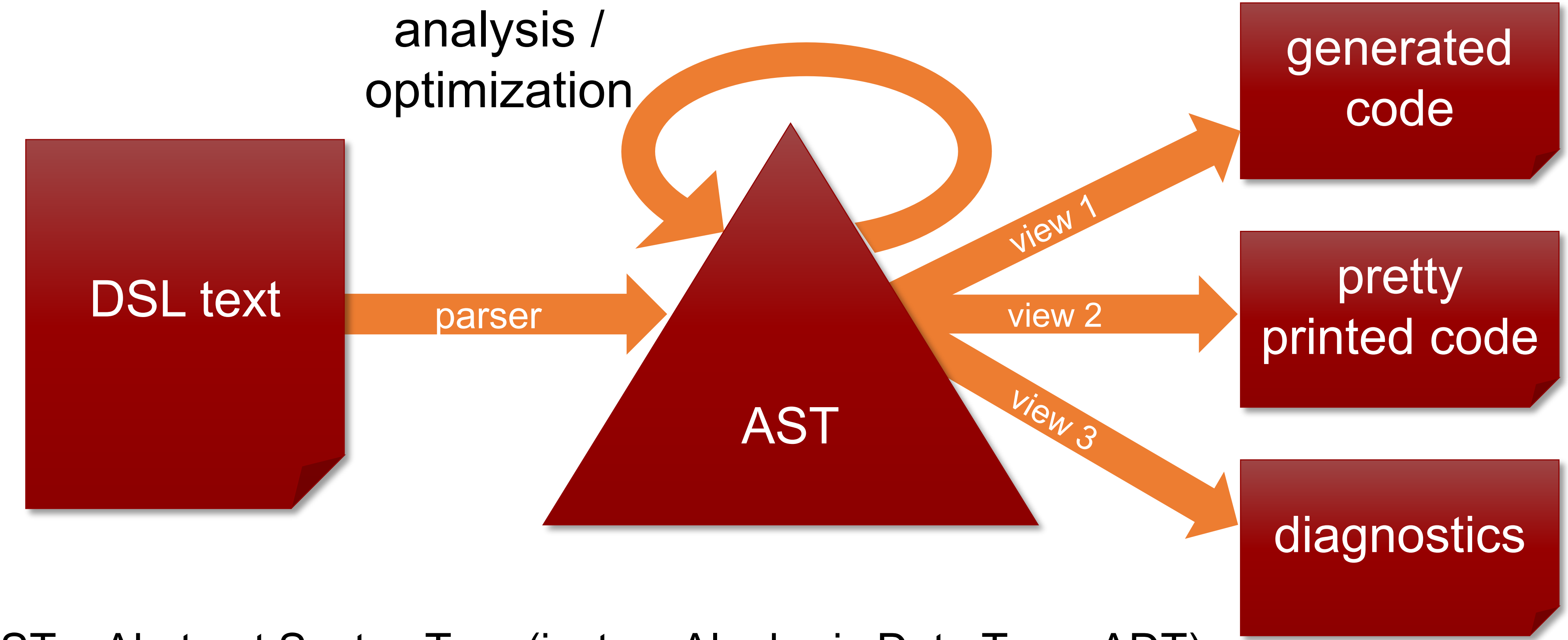
Embedded DSL, eDSL

- DSL embedded in another programming language
- No sharp border between DSL and library
  - DSL: computer language specialized to a particular application domain
    e.g. TOP, embedded SQL, JQuery, React, …
  - library: a collection of implementations of behaviour, that has a well-defined interface
  - we focus on techniques to implement eDSLs
- Tools for creating DSLs:
  - [JetBrains MPS] is a Java-based tool for designing DSLs
  - [Xtext] is a framework for developing PLs and DSLs
  - [Racket] is a toolchain designed to create DSLs and PLs
  - many features of Haskell/Clean are tailor-made for eDSLs

eDSL requirements

- Suited for the purpose
  - domain specific enough
  - nice syntax (whatever that may mean)
- Strongly typed
  - programming is hard enough, let a strong type system help us
  - a program that type checks in the host language should not contain DSL errors
- Extensible
  - add constructs to the language without the need to change existing code, e.g. a combinator
- Multiple interpretations is often convenient
  - evaluate, code generation, pretty print, optimize, analysis (like types, termination, …)
  - called **views** in the DSL community
- Safe and well typed variables

Compiler design versus deep embedding



- AST = Abstract Syntax Tree (just an Algebraic Data Type, ADT)
- deep embedded DSL = directly construct the AST in the host language
- no parser or separate source text

Running example

- Consider the very simple language WHILE[1]

  v a variable

  n a number

  $a \ = n \mid v \mid a + a \mid a - a \mid a * a$

  $b \ = \mathsf{TRUE} \mid \mathsf{FALSE} \mid a = a \mid a < a \mid \neg b \mid b \wedge b$

  $s \ = v := a \mid \texttt{skip} \mid s \,;\, s \mid \texttt{if}\ (\ b\ )\ \texttt{then}\ (\ s\ )\ \texttt{else}\ (\ s\ ) \mid \texttt{while}\ (\ b\ )\ (\ s\ )$

- Example: a statement to compute factorial of 4:

```
n := 4;
r := 1;
while (1 < n) (
   r := r * n ;
   n := n - 1
)
```

[1] see Nielson & Nielson: semantics with applications [1992]

Represent DSL by ADT: deep embedding

the grammar

```
a
      =    n
      |    v
      |    a + a
      |    a - a
      |    a * a
```

priority should be fixed by additional grammar rules

dot to avoid name conflicts

the data type

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

infix constructor with binding power

- we can add (if you dislike the +.)

```
instance  + AExpr
   where (+) a b = a +. b
```

The state in semantics

- To evaluate expressions we need to know the value of variables
- Store values in a function called **state**: *state* : *Variable* $\rightarrow$ *Z*
- The state can be updated: $[x \longmapsto v]\ s$ is the state that maps variable x to value v and all other variables to the value in s:

$$([x \longmapsto v]\ s)\ x = v$$
$$([x \longmapsto v]\ s)\ y = s\ y, \text{ if } x \neq y$$

- This is the notation used by Nielson & Nielson, we mimic this
- Next lecture we consider optimizations

The state

assigns a meaning to correct programs

errors in DSL programs are typically a fact of life

semantics

State : Variable $\rightarrow$ Z

- read is function application

- updates modifies function

$([x \longmapsto v]\ s)\ x = v$

$([x \longmapsto v]\ s)\ y = s\ y$, if $x \neq y$

DSL

```
:: State :== Var -> Int

emptyState :: State
emptyState = \x = 0

(|->) infix :: Var Int -> State -> State
(|->) x v = \s y = if (y == x) v (s y)
```

No declaration needed, any variable has a value. Fine in semantics, in a DSL we should check

equivalent:

```
\s =  (a, s)
\s .  (a, s)
\s -> (a, s)
```

The semantics of arithmetic expressions

- Use Scott brackets, ⟦ and ⟧, to indicate a pattern match on syntax elements in an **operational semantics**

$A : a \rightarrow State \rightarrow Z$

$A ⟦ n ⟧ s = N ⟦ n ⟧$ number

$A ⟦ v ⟧ s = s\ v$ variable

$A ⟦ a_1 + a_2 ⟧ s = A ⟦ a_1 ⟧ s + A ⟦ a_2 ⟧ s$

$A ⟦ a_1 - a_2 ⟧ s = A ⟦ a_1 ⟧ s - A ⟦ a_2 ⟧ s$

$A ⟦ a_1 * a_2 ⟧ s = A ⟦ a_1 ⟧ s \times A ⟦ a_2 ⟧ s$

The eval function is called A in semantics

syntax

mathematical operation

Semantic functions for arithmetic expressions

very similar for `BExpr` and `Stmt`

Scott brackets

$A : a \to State \to Z$

$A [\![n]\!]\, s \quad = N [\![n]\!]$

$A [\![v]\!]\, s \quad = s\ v$

$A [\![x+y]\!]\, s \quad = A [\![x]\!]\, s + A [\![y]\!]\, s$

$A [\![x-y]\!]\, s \quad = A [\![x]\!]\, s - A [\![y]\!]\, s$

$A [\![x*y]\!]\, s \quad = A [\![x]\!]\, s \times A [\![y]\!]\, s$

DSL, Clean as host language

```
A :: AExpr State -> Int
A (Int  n) s = n
A (Var  v) s = s v
A (x +. y) s = A x s + A y s
A (x -. y) s = A x s - A y s
A (x *. y) s = A x s * A y s
```

here we need `*.`

WHILE language design: deep embedding

- Host language does all it can to ensure type correctness in WHILE
  - using a different ADT for every type in the DSL
  - hard to extend, no overloading, no checking of variables

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

```
:: BExpr
 = TRUE
 | FALSE
 | (=.)   infix   4 AExpr AExpr
 | (<.)   infix   4 AExpr AExpr
 |  ~.              BExpr
 | (/\.) infixr 3 BExpr BExpr
```

```
:: Stmt
 = (:=.) infix  2 Var AExpr
 | (:.)  infixr 1 Stmt Stmt
 | Skip
 | If    BExpr Then Stmt
               Else Stmt
 | While BExpr Stmt
:: Then = Then
:: Else = Else
```

Limitations of the ADT approach

```
:: AExpr
  = Int   Int
  | Var   Var
  | (+.) infixl 6 AExpr AExpr
  | (-.) infixl 6 AExpr AExpr
  | (*.) infixl 7 AExpr AExpr
:: BExpr
  = TRUE | FALSE
  | (=.) infix 4    AExpr AExpr
  | (<.) infix 4    AExpr AExpr
  |  ~.             BExpr
  | (/\.) infixr 3 BExpr BExpr
```

only variables of type `Int`

arguments of type `AExpr` ensure integer values

as a result we cannot compare Booleans

arguments of type `BExpr` ensure Boolean values

Overload equality attempt 1

```
:: Expr
 = Num    Int
 | Var    Var
 | TRUE
 | FALSE
 | Plus   Expr Expr
 | Not    Expr
 | And    Expr Expr
 | Eq     Expr Expr
:: Val = I Int | B Bool | ERROR
```

- Allows `Eq TRUE FALSE`
- but also `Plus (Num 7) FALSE` 😱 😱

- Runtime errors possible during evaluation

```
eval :: Expr State -> Val
eval e s = case e of
  Num i     = I i
  Var v     = s v
  TRUE      = B True
  FALSE     = B False
  Plus x y = case (eval x s, eval y s) of
               (I a, I b) = I (a + b)
               _          = ERROR
  Not x     = case eval x s of
               (B b)      = B (not b)
               _          = ERROR
  And x y   = case (eval x s, eval y s) of
               (B a, B b) = B (a && b)
               _          = ERROR
  Eq x y    = case (eval x s, eval y s) of
               (I a, I b) = B (a == b)
               (B a, B b) = B (a == b)
               _          = ERROR
```

state of type `Var -> Val`

Overload equality, attempt 2

- Type argument indicates result type

```
:: Expr a
 = Lit  a     // Int and Bool
 | Var  Var   // Int
 | Plus (Expr Int)  (Expr Int)
 | Not  (Expr Bool)
 | And  (Expr Bool) (Expr Bool)
 | E.b: Eq (Expr b) (Expr b)
```

- omit -, * etc for brevity

- Allows:

```
Eq (Lit True) (Lit False)
Eq (Lit 7)    (Lit 42)
```

- but evaluation is a problem

```
eval :: (Expr a) State -> a
eval e s
  = case e of
      Lit  a   = a
      Var  v   = s v
      Plus x y = eval x s + eval y s
      Not  x   = not (eval x s)
      And  x y = eval x s && eval y s
      Eq   x y = eval x s == eval y s
```

Generalized Algebraic Data Types (GADTs)

- Bimap:

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

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

- Tell compiler that result of `Plus` has type `Int` and result of `And` has type `Bool`

```
:: Expr a
 = Lit  a
 | Plus (Expr Int)  (Expr Int)
 | And  (Expr Bool) (Expr Bool)

eval :: (Expr a) State -> a
eval e s = case e of
  Lit  a   = a
  Plus x y = eval x s +  eval y s
  And  x y = eval x s && eval y s
```

```
:: Expr a
 = Lit  a
 | Plus (BM a Int)  (Expr Int)  (Expr Int)
 | And  (BM a Bool) (Expr Bool) (Expr Bool)

eval :: (Expr a) State -> a
eval e s = case e of
  Lit  a        = a
  Plus {ba} x y = ba (eval x s +  eval y s)
  And  {ba} x y = ba (eval x s && eval y s)
```

Overload equality, attempt 3

for Int and Bool

```
:: Expr a
 = Lit    a
 | Var    (BM a Int)  Var
 | Plus   (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)
```

add a BM if the result type is not a

add class restrictions needed in all views

WHILE in GADTs: syntactic sugar

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

just add the identity bimap whenever needed

```
instance +   (Expr Int) where (+) x y = Plus bm x y
instance one (Expr Int) where one     = Lit 1

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

restrictions for all views needed

WHILE in GADTs: factorial statement

```
facStmt
  = "n" :=. num 4 :.
    "r" :=. one   :.
    While (num 1 <. var "n")
    (   "r" :=. var "r" * var "n" :.
        "n" :=. var "n" - one
    )
```

- this looks familiar
- all bimap/GADT magic hidden

WHILE in GADTs: showing expressions

```
class show a :: a [String] -> [String]

instance show (Expr a) | toString a where
  show e output = case e of
    Lit      a    = [toString a : output]
    Var   bm v    = [v          : output]
    Plus  bm x y  = ["(" : show x ["+"  : show y [")" : output]]]
    Not   bm x    = ["(" ,          "Not " : show x [")" : output]]
    And   bm x y  = ["(" : show x ["&&" : show y [")" : output]]]
    Eq    bm x y  = ["(" : show x ["==" : show y [")" : output]]]
```

- Bimap is not used because we do not produce a result of type a

WHILE in GADTs: showing expressions

```
class show a :: a [String] -> [String]

instance show (Expr a) | toString a where
  show e output = case e of
    Lit     a   = [toString a : output]
    Var  _  v   = [v          : output]
    Plus _  x y = ["(" : show x ["+"  : show y [")" : output]]]
    Not  _  x   = ["(" ,         "Not " : show x [")" : output]]
    And  _  x y = ["(" : show x ["&&" : show y [")" : output]]]
    Eq   _  x y = ["(" : show x ["==" : show y [")" : output]]]
```

- Bimap is not used because we do not produce a result of type a

WHILE in GADTs: evaluating expressions

```
eval :: (Expr a) State -> a
eval e s = case e of
  Lit  a         = a
  Var  {ba} v    = ba (s v)
  Plus {ba} x y  = ba (eval x s + eval y s)
  Not  {ba} x    = ba (not (eval x s))
  And  {ba} x y  = ba (eval x s && eval y s)
  Eq   {ba} x y  = ba (eval x s == eval y s)
```

Int

Bool

- Bimap is needed in almost every alternative
  - this is exactly the reason to introduce it
  - looks like a simple addition to the compiler, but in general it is harder

WHILE in GADTs: statements

```
:: Stmt
 = (:=.) infix  2 Var (Expr Int)
 | (:. ) infixr 1 Stmt Stmt
 | Skip
 | If    (Expr Bool) Then Stmt Else Stmt
 | While (Expr Bool) Stmt
:: Then = Then
:: Else = Else
```

Int

the desired static type checks

Bool

why not `Stmt a` and bimaps?

- There are no type arguments here
  - we do not need GADTs
  - we use GADTs to determine the required expression types

WHILE in GADTs: expression optimisation

```
opt :: (Expr a) -> Expr a
opt (Plus bm x y)
 = case (opt x, opt y) of
     (Lit n, Lit m) = bm.tba (Lit (n + m))
     (Lit 0, y    ) = bm.tba y
     (x,     y    ) = Plus bm x y
opt e = e
```

bimap: a ↔ Int

we need to transform `Expr Int` to `Expr a`

- We need a more complex bimap:

```
:: BM a b
 = { ab  :: a -> b,             ba  :: b -> a
   , tab :: A.t:(t a) -> t b, tba :: A.t:(t b) -> t a
   }
```

other examples need even more complex bimaps ☹

1 `BM` for all types ☺

GADTs á la Haskell: function types for constructors

```
:: Expr a
 = Lit    a
 | Var    (BM a Int)  Var
 | Plus   (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
```

experimental feature in Clean

```
:: Expr a
 = Lit    a                             -> Expr a
 | Var    Var                           -> Expr Int
 | Plus   (Expr Int)  (Expr Int)        -> Expr Int
 | Not    (Expr Bool)                   -> Expr Bool
 | And    (Expr Bool) (Expr Bool)       -> Expr Bool
 | E.b:Eq (Expr b)    (Expr b) & ==, toString b -> Expr Bool
```

in applications we do not need a bm ☺, determining the right bm is tricky ☹

Fixing Variable Definitions

works only for DSL without changing state (e.g. assignments)

HOAS: Higher-Order Abstract Syntax

- Key idea: use functions in the host language
- For the Lambda Calculus:

```
:: Lambda
= AbsL (Lambda -> Lambda)
| AppL Lambda Lambda
| AddL Lambda Lambda
| IntL Int
```

or a runtime type error

no state passed!
Clean does the hard work

```
evalL :: Lambda -> Lambda
evalL (AddL x y)
 = case (evalL x, evalL y) of
    (IntL n, IntL m) = IntL (n + m)
    (n, m)           = AddL n m
evalL (AppL f x)
 = case evalL f of
    AbsL f = evalL (f (evalL x))
    e      = AppL e (evalL x)
evalL e = e
```

```
idE    = AbsL \x = x
incE   = AbsL \x = AddL x (IntL 1)
twiceE = AbsL \f = AbsL \x = AppL f (AppL f x)
e1     = AppL (AppL twiceE (AppL idE incE)) (IntL 0)
```

we can check the types with a type argument

HOAS: Higher-Order Abstract Syntax

- How to print expressions?

```
:: Lambda
 = AbsL (Lambda -> Lambda)
 | AppL Lambda Lambda
 | AddL Lambda Lambda
 | IntL Int
 | VarL String
```

should not be used otherwise

- What to use as argument for printing?

```
varL v = VarL ("v" <+ v)
```

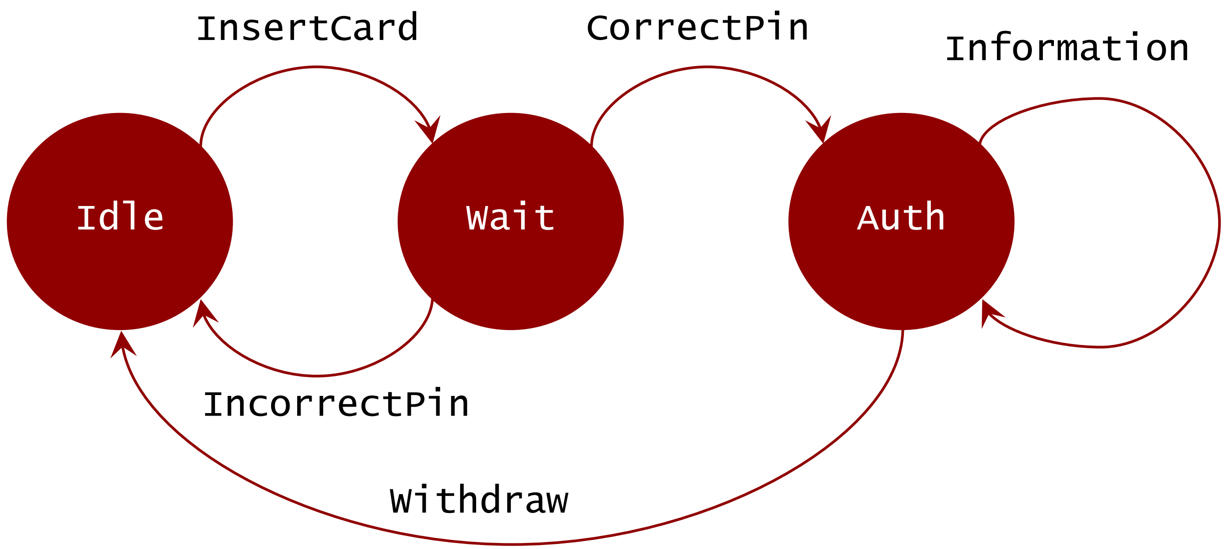
- Can be fixed with a parameter: PHOAS

```
print :: Lambda Int -> String
print (IntL x)   v = toString x
print (AddL x y) v = "(AddL " <+ print x v <+ " " <+ print y v <+ ")"
print (AppL f x) v = "(AppL " <+ print f v <+ " " <+ print x v <+ ")"
print (AbsL f  ) v = "(AbsL " <+ print (f (varL v)) (v+1) <+ ")"
print (VarL v)   n = v
```

A State Machine DSL With GADT

Case study

ATM

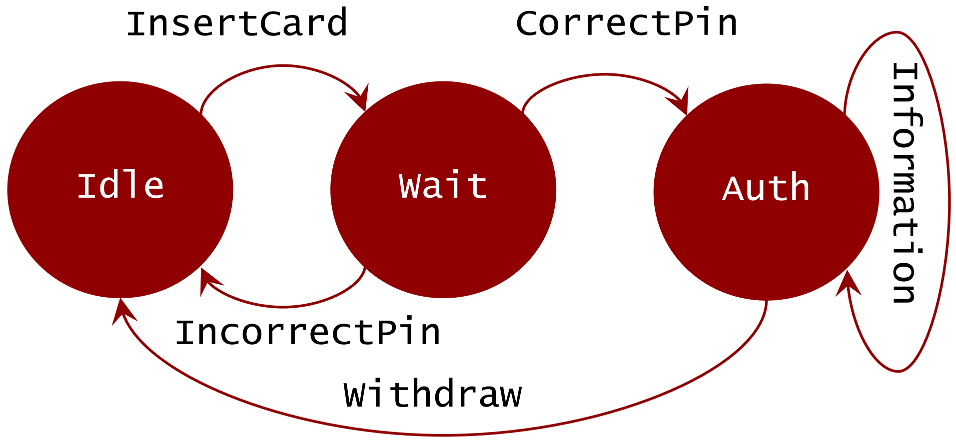


This allows traces such as:
`[InsertCard, Withdraw 36]` ☹

- Naive DSL implementation:

```
:: Trans = InsertCard | CorrectPin | IncorrectPin | Information | Withdraw Int
:: State = Idle | Wait | Auth
:: DSL :== [Trans]
```

ATM: GADT to the rescue



- Add arguments for initial and final state

```
:: Trans a b
   = InsertCard   (BM a Idle) (BM b Wait)
   | CorrectPin   (BM a Wait) (BM b Auth)
   | IncorrectPin (BM a Wait) (BM b Idle)
   | Information  (BM a Auth) (BM b Auth)
   | Withdraw (BM a Auth) (BM b Idle) Int
:: Idle = Idle
:: Wait = Wait
:: Auth = Auth
```

the constructors are never used

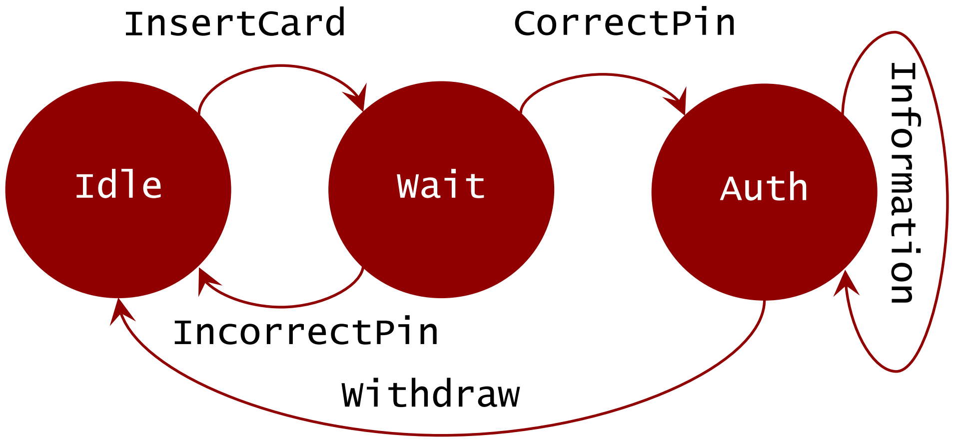
- Hide bimap with definitions such as:

```
insertCard = InsertCard bm bm
correctPin = CorrectPin bm bm
```

`[insertCard, withdraw 36]`:
cannot unify [Trans Idle Wait] with [Trans Auth Idle] ☺

but also useful traces are a type error ☹
how to fix this?

ATM: composing transitions



- List of transitions is type error, we need sequences

```
:: Trans a b
   = InsertCard   (BM a Idle) (BM b Wait)
   | CorrectPin   (BM a Wait) (BM b Auth)
   | IncorrectPin (BM a Wait) (BM b Idle)
   | Information  (BM a Auth) (BM b Auth)
   | Withdraw (BM a Auth) (BM b Idle) Int
   | E.c:(:.) infixl 1 (Trans a c) (Trans c b)

t1 :: Trans Idle Idle // success ☺
t1 = insertCard  :.
     correctPin  :.
     information :.
     withdraw 137
```

type system still rejects
`insertCard :. withdraw 7`

- highly extended version known as session types

ATM: additional transitions

- How to add skip and reset?
  - skip should not change state
  - reset should turn any state in `Idle`

```
:: Trans a b
  = InsertCard    (BM a Idle) (BM b Wait)
  | CorrectPin    (BM a Wait) (BM b Auth)
  | IncorrectPin (BM a Wait) (BM b Idle)
  | Information  (BM a Auth) (BM b Auth)
  | Withdraw      (BM a Auth) (BM b Idle) Int
  | E.c:(:.) infixl 1 (Trans a c) (Trans c b)
  | Skip          (BM a b)
  | Reset         (BM b Idle)
```

introduce a BM for any equality on types

Discussion

- Deep embedding: DSL = ADT
  - multiple views (evaluation, optimization, printing, …)
  - strong typing, no overloading, variable definition not checked
- GADT = Generalized Algebraic Data Types
  - allows strong typing and overloading
  - extending the DSL is still a problem: update all views
  - poor man's implementation with bimaps shows what is going on
  - ongoing research to find the optimal version of GADTs
- Higher-Order Abstract Syntax: HOAS
  - eliminates the need for a state (not for DSL with assignments, changing state)
  - fixes some problems with variables, but introduces new challenges