Teaching the Construction of Domain Specific Languages

Pieter Koopman & Rinus Plasmeijer

Radboud University, The Netherlands

TFPIE 2014

the context of Teaching the Construction of DSLs

- we teach advanced functional programming in master CS
- topics:
  - generic programming
  - GADTs
  - monads
  - task oriented programming: iTask
  - testing: Gast / QuickCheck
  - language semantics and implementation, ..
- department requires a focus on Domain Specific Languages
  - embedding a DSL in a FPL is well known
  - how does this effects our course?

new organisation of AFP

- one simple language as running example DSL for implementation strategies
  - While [Neilson and Neilson '92]

    a ::= n | x | a_1 + a_2 | a_1 * a_2 | a_1 - a_2

    b ::= **true** | **false** | a_1 = a_2 | $a_1 \leq a_2$ | $\neg b$ | $b_1 \wedge b_2$

    S ::= x **:=** a | **skip** | S_1 **;** S_2 | **if** b **then** S_1 **else** S_2

    | **while** b **do** S
  - advantages: small, but not too small, no direct fit to FPL
- introduce FPL constructs to implement such a DSL
  - monads, type classes, GADTs, generic programming, …
- stress DSL aspect in other topics + use them for While
  - task oriented programming: iTask + simulation of While
  - model-based testing: Gast + semantic properties of While

deep embedding of expressions

the grammar

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

dot to avoid name conflicts

priority should be fixed by additional grammar rules

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

semantic functions for arithmetic expressions

Scott brackets

$\mathcal{A} : a \rightarrow State \rightarrow Number$

$\mathcal{A}\ [[n]]\ s = \mathcal{N}\ [[n]]$

$\mathcal{A}\ [[v]]\ s = s\ v$

$\mathcal{A}\ [[a_1+a_2]]\ s = \mathcal{A}\ [[a_1]]\ s + \mathcal{A}\ [[a_2]]\ s$

$\mathcal{A}\ [[a_1-a_2]]\ s = \mathcal{A}\ [[a_1]]\ s - \mathcal{A}\ [[a_2]]\ s$

$\mathcal{A}\ [[a_1*a_2]]\ s = \mathcal{A}\ [[a_1]]\ s \times \mathcal{A}\ [[a_2]]\ s$

similar for Bexpr and semantic functions

DSL

```
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
```

see Nielson & Nielson 1992 only the syntax is improved

the state

semantics

$State : Variable \rightarrow Integer$

- read is function application

- updates modifies function

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

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

close to semantics, but inefficient

an explicit read instead of function application allows other implementations

DSL

```
State :: Var → Int

emptyState :: State
emptyState = λ x → 0

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

other views on expressions

- for a deep embedding it is easy to implement other manipulations of the DSL
  - e.g. pretty printing, collecting variables, optimization, ..
- collecting variables in `While`:

```
class vars a :: a -> [Var]

instance vars Stmt where
  vars (x :=. e)   = [x]
  vars (s :. t)    = vars s + vars t
  vars Skip        = []
  vars (IF c t e)  = vars t + vars e
  vars (While c b) = vars b
```

a simple simulator for While in iTasks

```
:: Val = { variable :: Var, value :: Int}

simulate :: Stmt -> Task Stmt
simulate stmt
 =(    viewInformation "values" []
         [ {variable = var, value = state var}
         \\ var <- vars stmt]
  ||-  updateInformation "current statement" [] stmt
  )
  >>* [OnAction ActionOk (hasValue simulate)]
where  state = ns stmt emptyEnv
```

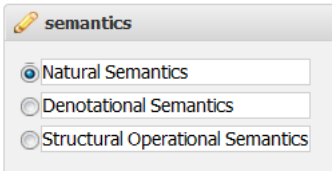
- add the required derives of `iTask` for the types in `While`

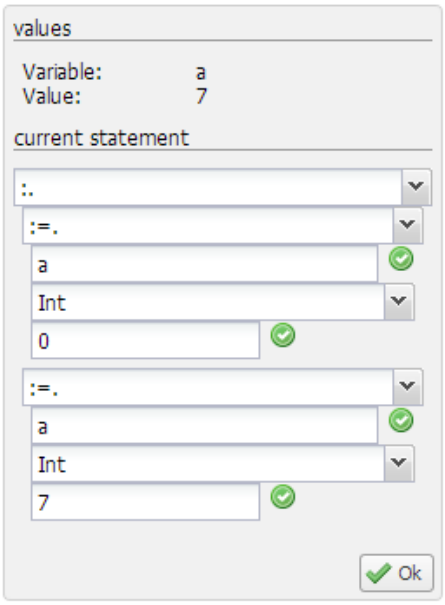
screen shot of simulation in iTasks

- expression

```
"a" :=. Int 0 :.
"a" :=. Int 7
```

- simulation allows validation of semantics
- here it shows that update really works





testing semantic properties

```
propFac :: (Stmt State -> State) -> Bool
propFac sem = sem facStmt emptyState "y" == 24

propFacAll :: Property
propFacAll = propFac For [ns, ds, sos ]

propSemiColon :: Stmt Stmt Stmt -> Property
propSemiColon s t u
 = eqEnv    (ns (s :. (t :. u)) emptyEnv)
            (ns ((s :. t) :. u) emptyEnv)
            (vars (s :. t :. u))
```

∀ sem

∀ sem ∈ { ns, ds, sos }

∀ s,t,u . s :. (t :. u) = (s :. t) :. u

- requires the generation of terminating statements

hiding the state in a monad

direct implementation

```
State :: Var → Int

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
```

semantic implication:
- order of evaluation
- expressions can change the state

monadic version

```
:: Sem :== State → (Int,State)

Sa :: AExp -> Sem
Sa (Int i)  = rtrn i
Sa (Var v)  = read v
Sa (x +. y) =
   Sa x >>= \n.
   Sa y >>= \m. rtrn (n + m)
Sa (x -. y) =
   Sa x >>= \n.
   Sa y >>= \m. rtrn (n - m)
```

mixing expressions in While

- While has separate arithmetic and Boolean expressions
  - host language can check types in While ☺
  - imposes (too) much restrictions on expression
    - no Boolean variables, no equality on Booleans, ..
- these restrictions disappear in a single expression type

```
:: Val = I Int | B Bool
:: Exp = Val Val
       | Var Var
       |  (+.)    infixl 6  Exp Exp
       |  (&&.)   infixl 6  Exp Exp
       |  (==.)   infixr 3  Exp Exp
       | ..
```

- but now we can have runtime type errors ☹

poor man GADTs

- GADTs are designed to solve these problems
- Clean does not have GADTS, but we can mimic them

```
:: BM a b = { t :: a → b, f :: b → a }      // bimap

bm :: BM a a
bm = {f = id, t = id}

:: Expr a
   = Num   (BM a Int) Int
   | Var   (BM a Int) Var
   | TRUE  (BM a Bool)
   | 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
   | …
```

poor man GADTs 2

```
num  = Num bm
var  = Var bm
true = TRUE bm
instance + (Expr Int) where (+) x y = Plus bm x y
```

- showing these expressions

```
show :: (Expr a) -> [String] | toString a
show (Num _ i)     = [toString i: s]
show (Var _ v)     = [v: s]
show (Plus  _ x y) = ["(":show x]+["+":show y]+[")"]
```

- evaluation

```
eval :: (Expr a) State -> a | == a
eval (Num   {f} i)   s = f i
eval (Var   {f} v)   s = f (read v s)
eval (Plus  {f} x y) s = f (eval x s + eval y s)
eval (TRUE  {f})     s = f True
```

Int

Bool

shallow embedding of While

```
:: AShllw :== State -> Int

int :: Int -> AShllw
int i = \s. i

var :: Var -> AShllw
var v = \s. s v

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

Start = (int 6 * var "x") sx
sx = ("x" |-> 7) s0
```

- solves many type issues ☺
- there is only one view! ☹

multiple views in shallow embedding

```
class E e where
  val            :: Val  -> e
  var            :: V    -> e
  (+.) infixl 6  :: e e  -> e
  (=.) infix  2  :: V e  -> e // assignment
  …
```

- Showing

```
instance E [String] where
  val  v   = [toString v]
  var  v   = [v]
  (+.) x y = ["(":x] ++ [" +. ": y] ++ [")"]
  (=.) v e = [v," =. ": e]
```

- evaluation

```
instance E Sem where
  val  v   = rtn v
  var  v   = read v
  (+.) x y = x >>= \a. y >>= \b. rtn (a + b)
  (=.) v e = e >>= write v
```

16

other DSLs in the course: ITasks

- task oriented programming
  - a monadic DSL for tasks
  - generic generation of web-interface

```
class iTask a
  | gEditor{|*|}, gEditMeta{|*|} , .., gEq{|*|}, TC a

(>>=) infixl 1 :: (Task a) (a -> Task b) -> Task b
                                    | iTask a & iTask b
(>>*) infixl 1 :: (Task a) [TaskStep a b] -> Task b
                                    | iTask a & iTask b
return :: a -> Task a | iTask a
```

- we teach how to use this DSL and reveal the construction of this DSL

other DSLs in the course: testing

- in model-based testing the model is expressed in a DSL
  - logical properties

```
class TestArg a | genShow{|*|}, ggen{|*|} a
class Testable a :: a RandomStream Admin -> [Admin]
instance Testable Bool
instance Testable (a->b) | Testable b & TestArg a
```

- conformance of state machines

```
:: Spec state input output
       :== state input -> [Trans output state]
:: Trans output state
       = Pt [output] state | Ft ([output]->[state])
```

- we teach how to use this DSL and reveal the construction of this DSL

other teaching opportunities with DSLs

- a functional DSL
- consequences of lazy evaluation
- other well-known DSLs
  - parser combinators
- efficiency of DSL
- handling nontermination
- program optimization
- type checking
- the type tells (almost) everything about DSL constructs
- why we often use a shallow embedded DSL with one view
- ...

opinion of students

- most of the students (>90%) really like the course
  - the see the advantages of a single example
  - due to our focus on multiple views they tend to prefer deep embedding of a DSL
  - can use DSLs and implement them in different ways

- a small minority has many remarks
  - we know everything about testing
  - we know everything about semantics
  - proving properties of monads is more interesting
  - Haskell is a better language
  - Object Orientation is better than this functional stuff
  - nobody in the world will ever use a FPL for something useful

take home message

- the enforced focus on DSLs improved our Advance Functional Programming course
  - embedded DSLs are a perfect fit for FPL
  - DSLs are an excellent motivation for advanced FPL concepts
  - course can be improved by a more systematic focus on DSLs
- a running example helps to compare the approaches
  - deep embedding
  - GADT
  - shallow embedding
  - class based shallow embedding
  - While has the right size and shape for this role