Certified Software Development with Dependent Types in Idris

Lecture 11. Defining EDSLs

Vitaly Bragilevsky (bravit@sfedu.ru)

I.I. Vorovich Institute of Mathematics, Mechanics and Computer Science
Southern Federal University, Rostov-on-Don, Russia
part of Erasmus+ teaching mobility agreement with University of Twente

Definitions

- DSL — domain-specific language — is a computer language specialized to a particular application domain (*in contrast with general purpose language*).
- EDSL — embedded DSL — implemented as libraries which exploit the syntax of their host language or a subset thereof, while adding domain-specific language elements (data types, routines, methods, macros etc.) — parts of object language.
- Defining EDSL does not neccessary mean extending host language syntax, but sometimes it is useful in order to get rid of unspecific details (such as type information e.g.).

Support for EDSLs in Idris

- Implementing domain entities via Idris type system.
- Extending do-notation.
- Defining new syntax rules.
- Overloading host language syntax for use by object language (limited to variable bindings).

Содержание

Content

Writing Papers: Domain Entities

- Writing papers.
- Submitting/reviewing process.
- It should be impossible to submit same paper twice or to reject paper that was not previously submitted.
- We will not try to implement this process (it's not an AI class afterall!).
- Instead we will model it.

PaperState and Paper

```
data PaperState =  Written | Reviewing | Accepted | Rejected

data Paper : PaperState -> Type where
  MkPaper : Paper s
```

Paper Events

- writing
- submitting
- reviewing
- rejecting
- revising

PaperEvent

```
data PaperEvent : Type -> Type where
  Write   :                          PaperEvent (Paper Written)
  Submit  : Paper Written   -> PaperEvent (Paper Reviewing)
  Accept  : Paper Reviewing -> PaperEvent (Paper Accepted)
  Reject  : Paper Reviewing -> PaperEvent (Paper Rejected)
  Revise  : Paper Rejected  -> PaperEvent (Paper Reviewing)
```

What Is a Language Here?

- We have actions and we want to build them in a sequence.
- Sounds like a monad (or at least do-notation).

```
data PaperLang : Type -> Type where
  Action : PaperEvent a -> PaperLang a
  (>>=) : PaperLang a -> (a -> PaperLang b) -> PaperLang b
```

- Nothing is a function here, only data constructors!

Example Scripts

```
prog1 : PaperLang (Paper Accepted)
prog1 = Action Write >>= Action . Submit >>= Action . Accept

prog2 : PaperLang (Paper Accepted)
prog2 = do
   p <- Action Write
   p <- Action (Submit p)
   p <- Action (Reject p)
   p <- Action (Revise p)
   Action (Accept p)
```

papers.idr: let's cheat a little bit!

What Is prog2 Technically?

```
Idris> :printdef prog2
prog2 : PaperLang (Paper Accepted)
prog2 = ((Action Write) >>=
        (\p =>
           ((Action (Submit p)) >>=
            (\p5 =>
               ((Action (Reject p5)) >>=
                (\p8 => ((Action (Revise p8)) >>=
                    (\p11 => Action (Accept p11)))))))))
```

- It's some piece of data build via (>>=) data constructor.

Little Improvement: Implicit Functions

```
implicit
action : PaperEvent a -> PaperLang a
action = Action

prog2' : PaperLang (Paper Accepted)
prog2' = do
   p <- Write
   p <- Submit p
   p <- Reject p
   p <- Revise p
   Accept p
```

- Implicit function action will be called automatically to satisfy type checking.

Introducing New Keywords

```
syntax write = Action (Write)
syntax submit = \p => Action (Submit p)
syntax accept = \p => Action (Accept p)
syntax reject = \p => Action (Reject p)
syntax revise = \p => Action (Revise p)
syntax AcceptedPaper = PaperLang (Paper Accepted)
```

```
prog3 : AcceptedPaper
prog3 = write >>= submit >>= accept

prog4 : AcceptedPaper
prog4 = write >>= submit >>= reject >>= revise >>=
                             reject >>= revise >>= accept
```

Content

Strange Things in Do-blocks

```
sum : Int
sum = do
        15
        15
        -5
        19
        -2
```

```
(>>=) : Int -> (Int -> Int) -> Int
(>>=) n f = n + f 0
```

```
Idris> sum
42 : Int
```

```
(>>=) : String -> (String -> String) -> String
(>>=) n f = n ++ f ""

sum : String
sum = do
        "15"
        "10"
```

```
(>>=) : String -> (String -> List String) -> List String
(>>=) n f = n :: f ""
syntax END = []

sum2 : List String
sum2 = do
         "10"
         "20"
         "30"
         END
```

Content

```
syntax CALL [f] ON [t] WITH [a] = f t a;

g : Int -> Int -> IO ()
g a b = printLn $ a + b

h : String -> Bool -> IO ()
h s False = printLn s
h s True = printLn ""

main : IO ()
main = do
    CALL g ON 10 WITH 5
    CALL g ON 1 WITH 3
    CALL h ON "QQ" WITH False
```

- Introducing new keywords.
- Transforming expressions with new keywords to function calls.

Other Examples

```
syntax [var] ":=" [val]                  = Assign var val
syntax [test] "?" [t] ":" [e]            = if test then t else e
syntax select [x] from [t] "where" [w] = SelectWhere x t w
syntax select [x] from [t]               = Select x t
```

- Keywords and symbols in quotation marks.

Explicit Loops

```
syntax for {x} "in" [xs] ":" [body] = for xs (\x => body)

main : IO ()
main = do for x in [1..10]:
              putStrLn ("Number " ++ show x)
          putStrLn "Done!"
```

```
main : IO ()
main = do for x in [1..10]:
              do {putStr ("Number " ++ show x); putStrLn ""}
          putStrLn "Done!"
```

- Representing bound variables in {}.

```
for : (Traversable t,Applicative f) => t a -> (a -> f b) -> f (t b)
```

Content

Expressions and Types

```
e ::=
   n                        (number)
   x                        (variable)
   λx.e                     (lambda)
   e e                      (application)
   e ○ e                    (operation)
   if e then e else e       (conditional)
```

```
t ::=
   int
   bool
   t → t
```

Environment and Typing Context

- Environment contains values of variables.
- Typing context contains types of variables.

$$
\begin{aligned}
\Gamma \quad ::= & \\
& \varnothing \\
& \Gamma, x : T
\end{aligned}
$$

Typing Judgement and Typing Rules

$$\Gamma \vdash e : T$$

$$\frac{n - \text{number}}{\varnothing \vdash n : int} \quad (Val) \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (Var)$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \to T_2} \quad (Lam)$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2} \quad (App)$$

$$\frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash if\ \ e_1\ \ then\ e_2\ \ else\ \ e_3 : T} \quad (If)$$

$$\frac{\circ - \text{operation over } T_1 \text{ and } T_2 \text{ resulting in } T_3 \quad \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \circ e_2 : T_3} \quad (Op)$$

De Bruijn Indices

$$\lambda x.\lambda y.x(yx) \quad \Longrightarrow \quad \lambda.\lambda.1'(0'1')$$

$$x + y \quad \Longrightarrow \quad 0' + 1'$$
$$\text{where } \Gamma = [\ldots, int, int]$$

- DeBruijn indices directly correspond to the (reversed) position in the typing context and environment.

Types and Their Interpretation

```
data Ty = TyInt | TyBool | TyFun Ty Ty

interpTy : Ty -> Type
interpTy TyInt        = Int
interpTy TyBool       = Bool
interpTy (TyFun s t) = interpTy s -> interpTy t
```

Environment, Typing Context, and Search for Value

```
using (G : Vect n Ty)

  data Env : Vect n Ty -> Type where
      Nil  : Env Nil
      (::) : interpTy a -> Env G -> Env (a :: G)

  data HasType : (i : Fin n) -> Vect n Ty -> Ty -> Type where
      Stop : HasType FZ (t :: G) t
      Pop  : HasType k G t -> HasType (FS k) (u :: G) t

  lookup : HasType i G t -> Env G -> interpTy t
  lookup Stop    (x :: xs) = x
  lookup (Pop k) (x :: xs) = lookup k xs
  lookup Stop    [] impossible
```

- `HasType i G t` means exactly $\Gamma \vdash i' : t$, where i' is de Bruijn index.

Expressions

```
data Expr : Vect n Ty -> Ty -> Type where
  Var : HasType i G t -> Expr G t
  Val : (x : Int) -> Expr G TyInt
  Lam : Expr (a :: G) t -> Expr G (TyFun a t)
  App : Lazy (Expr G (TyFun a t)) -> Expr G a -> Expr G t
  Op  : (interpTy a -> interpTy b -> interpTy c) ->
        Expr G a -> Expr G b -> Expr G c
  If  : Expr G TyBool -> Expr G a -> Expr G a -> Expr G a
```

Interpreting Expressions

```
total
interp : Env G -> (e : Expr G t) -> interpTy t
interp env (Var i)      = lookup i env
interp env (Val x)      = x
interp env (Lam sc)     = \x => interp (x :: env) sc
interp env (App f s)    = (interp env f) (interp env s)
interp env (Op op x y) = op (interp env x) (interp env y)
interp env (If x t e)   = if interp env x
                            then interp env t
                            else interp env e
```

Testing

```
ef : Expr G (TyFun TyInt (TyFun TyInt TyInt))
ef = Lam (Lam (Op (+) (Var Stop) (Var (Pop Stop))))

e : Expr G TyInt
e = App (App ef (Val 5)) (Val 10)
```

```
Idris> interp [] e
15:int
```

Content

Overloading Syntax

```
lam_ : TTName -> Expr (a :: G) t -> Expr G (TyFun a t)
lam_ _ = Lam

dsl expr
    lambda = lam_
    variable = Var
    index_first = Stop
    index_next = Pop
```

```
eId : Expr G (TyFun TyInt TyInt)
eId = expr (\x => x)

eAdd : Expr G (TyFun TyInt (TyFun TyInt TyInt))
eAdd = expr (\x, y => Op (+) x y)
eDouble : Expr G (TyFun TyInt TyInt)
eDouble = expr (\x => App (App eAdd x) (Var Stop))
```

```
eFac : Expr G (TyFun TyInt TyInt)
eFac = expr (\x => If (Op (==) x (Val 0))
                  (Val 1)
                  (Op (*) (App eFac (Op (-) x (Val 1))) x))
```

```
testFac : Int
testFac = interp [] eFac 4

main : IO ()
main = printLn testFac
```

Quick review of res.idr and Resimp.idr.

Bibliography

1. The Idris Tutorial
   http://docs.idris-lang.org/en/latest/tutorial/index.html