Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ECI 2024

Deep Embedding de Lenguajes Tipados

Lenguajes Tipados - Deep Embedding

Volviendo al lenguaje con distintos tipos, pero ahora como deep embedding

```
data Expr :: *where
  Val    :: Int → Expr
  Add    :: Expr → Expr → Expr
  IsZero :: Expr → Expr
  If     :: Expr → Expr → Expr → Expr
```

Evaluador con type-checking

Evaluador parcial, o con type-checking dinámico

```
data Res = RI Int | RB Bool

eval :: Expr → Res
eval (Val n)      = RI n
eval (Add e1 e2) = case (eval e1, eval e2) of
                     (RI n1, RI n2) → RI (n1 + n2)
                     _              → error "type error: Add"
eval (IsZero e)   = case eval e of
                     RI n  → RB (n ≡ 0)
                     _     → error "type error: IsZero"
eval (If e1 e2 e3) = case eval e1 of
                     RB b →
                       case (eval e2, eval e3) of
                         (RI n2, RI n3)   → RI  $ if b then n2 else n3
                         (RB b2, RB b3) → RB $ if b then b2 else b3
                         _              → error "type error: If branches"
                     _     → error "type error: If condition"
```

Type-checking

Podría desacoplar el type-checking de la evaluación

```
data EType = TInt | TBool

infix 6 .::

(.::) :: Expr → EType → Bool
(Val n)      .:: TInt  = True
(Add e1 e2) .:: TInt  = e1 .:: TInt  ∧ e2 .:: TInt
(IsZero e)   .:: TBool = e  .:: TInt
(If c e1 e2) .:: t     = c  .:: TBool ∧ e1 .:: t ∧ e2 .:: t
_            .:: _     = False

wellTyped :: Expr → Bool
wellTyped e = e .:: TInt ∨ e .:: TBool
```

Type-checking

Podría desacoplar el type-checking de la evaluación

```
data EType = TInt | TBool

infix 6 .::

(.::) :: Expr → EType → Bool
(Val n)      .:: TInt  = True
(Add e1 e2) .:: TInt  = e1 .:: TInt  ∧ e2 .:: TInt
(IsZero e)   .:: TBool = e  .:: TInt
(If c e1 e2) .:: t     = c  .:: TBool ∧ e1 .:: t ∧ e2 .:: t
_            .:: _     = False

wellTyped :: Expr → Bool
wellTyped e = e .:: TInt ∨ e .:: TBool
```

y sólo evaluar en caso de tipar correctamente

```
safeEval :: Expr → Res
safeEval e | wellTyped e = eval e
           | otherwise   = error "type error"
```

Tagless - Deep Embedding (Initial)

Podemos usar el enfoque tagless para que el sistema de tipos del lenguaje anfitrión realice el type-checking

```
class ExprT (e :: * -> *) where
  valT    :: Int -> e Int
  addT    :: e Int -> e Int -> e Int
  isZeroT :: e Int -> e Bool
  ifT     :: e Bool -> e t -> e t -> e t
```

Tagless - Deep Embedding (Initial)

Podemos usar el enfoque tagless para que el sistema de tipos del lenguaje anfitrión realice el type-checking

```
class ExprT (e :: * -> *) where
  valT    :: Int -> e Int
  addT    :: e Int -> e Int -> e Int
  isZeroT :: e Int -> e Bool
  ifT     :: e Bool -> e t -> e t -> e t
```

Necesito empaquetar *Expr* en un $* \to *$ con un phantom type

```
newtype WExpr a = E Expr

instance ExprT WExpr where
  valT n                  = E (Val n)
  addT (E e1) (E e2)      = E (Add e1 e2)
  isZeroT (E e)           = E (IsZero e)
  ifT (E c) (E e1) (E e2) = E (If c e1 e2)
```

Intérpretes Tipados

Puedo definir un observador como pretty-printing sin problemas

```
ppExpr (E (Val n))      = show n
ppExpr (E (Add e1 e2)) = "(" ++ ppExpr (E e1) ++ " + " ++ ppExpr (E e2) ++ ")"
ppExpr (E (IsZero e))   = "isZero(" ++ ppExpr (E e) ++ ")"
ppExpr (E (If c e1 e2)) = "if " ++ ppExpr (E c) ++
                          " then " ++ ppExpr (E e1) ++
                          " else " ++ ppExpr (E e2)
```

Intérpretes Tipados

Puedo definir un observador como pretty-printing sin problemas

$$
\begin{aligned}
&ppExpr\ (E\ (Val\ n)) &&= show\ n \\
&ppExpr\ (E\ (Add\ e1\ e2)) &&= \texttt{"("} +\!\!+ ppExpr\ (E\ e1) +\!\!+ \texttt{" + "} +\!\!+ ppExpr\ (E\ e2) +\!\!+ \texttt{")"} \\
&ppExpr\ (E\ (IsZero\ e)) &&= \texttt{"isZero("} +\!\!+ ppExpr\ (E\ e) +\!\!+ \texttt{")"} \\
&ppExpr\ (E\ (If\ c\ e1\ e2)) &&= \texttt{"if "} +\!\!+ ppExpr\ (E\ c) +\!\!+ \\
& && \quad \texttt{" then "} +\!\!+ ppExpr\ (E\ e1) +\!\!+ \\
& && \quad \texttt{" else "} +\!\!+ ppExpr\ (E\ e2)
\end{aligned}
$$

Pero no puedo definir algo como

$$
\begin{aligned}
&evalExpr\ (E\ (Val\ n)) &&= n \\
&evalExpr\ (E\ (IsZero\ e)) &&= evalExpr\ (E\ e) \equiv 0 \\
&evalExpr\ (E\ (Add\ e1\ e2)) &&= evalExpr\ (E\ e1) + evalExpr\ (E\ e2) \\
&evalExpr\ (E\ (If\ c\ e1\ e2)) &&= \texttt{if}\ evalExpr\ (E\ c)\ \texttt{then}\ evalExpr\ (E\ e1) \\
& && \quad \texttt{else}\ evalExpr\ (E\ e2)
\end{aligned}
$$

Intérpretes Tipados

Puedo definir un observador como pretty-printing sin problemas

```
ppExpr (E (Val n))      = show n
ppExpr (E (Add e1 e2)) = "(" ++ ppExpr (E e1) ++ " + " ++ ppExpr (E e2) ++ ")"
ppExpr (E (IsZero e))   = "isZero(" ++ ppExpr (E e) ++ ")"
ppExpr (E (If c e1 e2)) = "if " ++ ppExpr (E c) ++
                          " then " ++ ppExpr (E e1) ++
                          " else " ++ ppExpr (E e2)
```

Pero no puedo definir algo como

```
evalExpr (E (Val n))      = n
evalExpr (E (IsZero e))   = evalExpr (E e) == 0
evalExpr (E (Add e1 e2)) = evalExpr (E e1) + evalExpr (E e2)
evalExpr (E (If c e1 e2)) = if evalExpr (E c) then evalExpr (E e1)
                                              else evalExpr (E e2)
```

```
Couldn't match expected type 'Int' with actual type 'Bool'
In the expression: evalExpr (E e) == 0
...
```

Intérpretes Tipados (2)

Tampoco puedo definir

$$
\begin{aligned}
&\mathit{evalExpr} :: \mathit{WExpr}\ t \to t \\
&\mathit{evalExpr}\ (E\ (\mathit{Val}\ n)) &&= n \\
&\mathit{evalExpr}\ (E\ (\mathit{IsZero}\ e)) &&= \mathit{evalExpr}\ (E\ e) \equiv 0 \\
&\mathit{evalExpr}\ (E\ (\mathit{Add}\ e1\ e2)) &&= \mathit{evalExpr}\ (E\ e1) + \mathit{evalExpr}\ (E\ e2) \\
&\mathit{evalExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) &&= \texttt{if}\ \mathit{evalExpr}\ (E\ c)\ \texttt{then}\ \mathit{evalExpr}\ (E\ e1) \\
&&&\quad\ \texttt{else}\ \mathit{evalExpr}\ (E\ e2)
\end{aligned}
$$

Intérpretes Tipados (2)

Tampoco puedo definir

$$
\begin{aligned}
&\mathit{evalExpr} :: \mathit{WExpr}\ t \to t \\
&\mathit{evalExpr}\ (E\ (\mathit{Val}\ n)) &&= n \\
&\mathit{evalExpr}\ (E\ (\mathit{IsZero}\ e)) &&= \mathit{evalExpr}\ (E\ e) \equiv 0 \\
&\mathit{evalExpr}\ (E\ (\mathit{Add}\ e1\ e2)) &&= \mathit{evalExpr}\ (E\ e1) + \mathit{evalExpr}\ (E\ e2) \\
&\mathit{evalExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) &&= \texttt{if}\ \mathit{evalExpr}\ (E\ c)\ \texttt{then}\ \mathit{evalExpr}\ (E\ e1) \\
& && \quad\ \texttt{else}\ \mathit{evalExpr}\ (E\ e2)
\end{aligned}
$$

```
Couldn't match expected type 't' with actual type 'Int'
't' is a rigid type variable bound by
  the type signature for:
    evalExpr :: forall t. WExpr t -> t
...
In the expression: n
  In an equation for 'evalExpr': evalExpr (E (Val n)) = n
...
```

Intérpretes Tipados (3)

Se puede definir un *workaround* usando type-clases

```
class Eval t where
  weval :: WExpr t → t

instance Eval Int where
  weval (E (Val n))      = n
  weval (E (Add e1 e2)) = weval (E e1) + weval (E e2)
  weval (E (If c e1 e2)) = if weval (E c) then weval (E e1) else weval (E e2)

instance Eval Bool where
  weval (E (IsZero e))   = (weval (E e) :: Int) ≡ 0
  weval (E (If c e1 e2)) = if weval (E c) then weval (E e1) else weval (E e2)
```