How to Implement Typed Language ~Tagless Final~

Syouki Tsuyama (Tokyo Tech)

My research

How to implement typed language in type-safe way

Use a dependently-typed language to
Implement the Intrinsically-typed interpreter(ITI)

⬇

Type-checking ITI = proof of type-safety

Use this approach for effectful, coeffectful languages.

Related work : Tagless Final

An approach to implementing a typed language
without advanced type systems(GADT, dependent types)

Need only type class (trait)

Built in Scala, Haskell, Ocaml

Contents

- Type Preserved Interpretation
- Tagless Final
- Relevance to my research (ITI vs Tagless)
- Appendix
  - Expression Problem
  - How to Bind Variables

Contents

- Type Preserved Interpretation
- Tagless Final
- Relevance to my research (ITI vs Tagless)
- Appendix
  - Expression Problem
  - How to Bind Variables

Implementing Language ~Initial Approach~

When using Haskell(no extension)

```haskell
-- BoolNat
data Expr =
  Num Int | Bl Bool |
  Add Expr Expr |
  If Expr Expr Expr

data Val =
  VNum Int | VBl Bool
```

```haskell
eval :: Expr -> Val
eval (Num n) = VNum n
eval (Bl b) = VBl b
. . .
```

Tagging values with types

Initial Approach ~Tagging Problem~

```
eval :: Expr -> Val
. . .
eval (If e1 e2 e3) =
  let (VBl b) = eval e1 in
  if b then eval e1 else eval e2
```

No guarantee that `e1` is not a number

Non-exhaustive pattern-match incur a peformance penalty

Type-preserved Interpretation

Expression of type T $\xrightarrow{\texttt{evaluate}}$ Value of type T

e.g. `If e1 e2 e3`

Guarantee to be evaluated to Bool
by type system of meta language

Several approaches : ITI, Tagless Final

Contents

- Type Preserved Interpretation
- Tagless Final
- Relevance to my research (ITI vs Tagless)
- Appendix
  - Expression Problem
  - How to Bind Variables

Another Approach : Tagless Final

An approach to implementing a typed language
without advanced type systems(GADT, dependent types)

Need only type class (trait)

Built in Scala, Haskell, Ocaml

BoolNat in Tagless Final

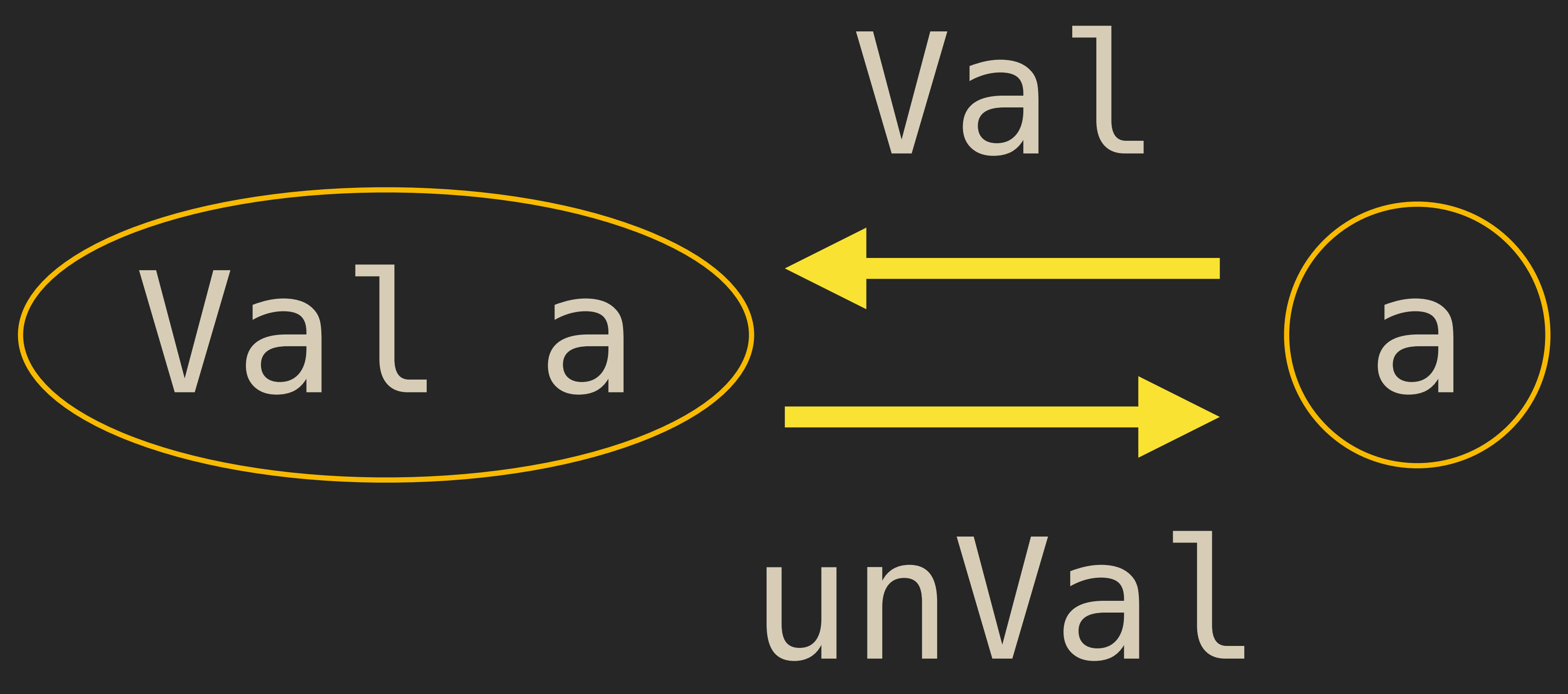
```
class Symantics exp where
  bool :: Bool -> exp Bool
  if   :: exp Bool -> exp a -> exp a -> exp a
  num  :: Int -> exp Int
  add  :: exp Int -> exp Int -> exp Int
```

Correspond to typing rules

$$\frac{\texttt{n : Nat} \qquad \texttt{m : Nat}}{\texttt{add n m : Nat}}$$

BoolNat Interpreter in Tagless Final

```haskell
newtype Val a =
  Val { unVal :: a }
```




```haskell
instance Symantics State s where
  num n = Val n
  add n m = Val (unVal n + unVal m)
```

Running Example

```
-- 1 + 2
e1 :: Val Int
e1 = add (num 1) (num 2)

e1
>> Val 3

-- ill-typed term (true + 1)
illtyped = add (bool True) (num 1)
☝️  type error!
```

Can be excluded by type checking

Contents

- Type Preserved Interpretation
- Tagless Final
- Relevance to my research (ITI vs Tagless)
- Appendix
  - Expression Problem
  - How to Bind Variables

An Approach : Intrinsically-typed interpreter

`Expr T` : Type for the terms of type T

```
data Expr : Ty → Set where
  true  : Expr Bool
  false : Expr Bool
  num   : ℕ → Expr Nat
  succ  : Expr Nat → Expr Nat
```

```
eval : Expr T → Val T
```

Type of Interpreter = Statement of type-safety

ITI vs Tagless Final

	GADT	DepType	TypeClass (Trait)
ITI	✓	✓	
Tagless			✓

ITI vs Tagless Final

GADT & Dependant Types

- Strict assuarance by proof assistant
- Available for more advanced type system requiring type-level calculation

Tagless Final

- Easy
- Available in currently popular languages
- Solution of “Expression Problem”

Expression Problem

```haskell
-- BoolNat
data Expr =
  Num Int | Bl Bool |
  Add Expr Expr |
  If Expr Expr Expr
```

Extend with List

```haskell
data Expr =
  . . . |
  Nil |
  Cons Expr Expr
```

Need to modify all functions using Expr (eval, size, etc.)

Solve with Tagless

New commands → New type class

New function clause → New instance

for only the fuctions
you want to extend

Extending in Tagless Final

New commands

```
class (Symantics exp) => ListSymantics exp where
  nil :: exp [a]
  cons :: exp a -> exp [a] -> exp [a]
```

Append interpreter

```
instance ListSymantics Val where
  nil = Val []
  cons x xs = Val (unVal x : unVal xs)
```

Effect

```haskell
instance Symantics (State s) where
  num n = return n
  add e1 e2 = do
    n <- e1
    m <- e2
    return $ n + m
  lam f = return $ f
  app e1 e2 = do
    f <- e1
    v <- e2
    f v
```

Binding