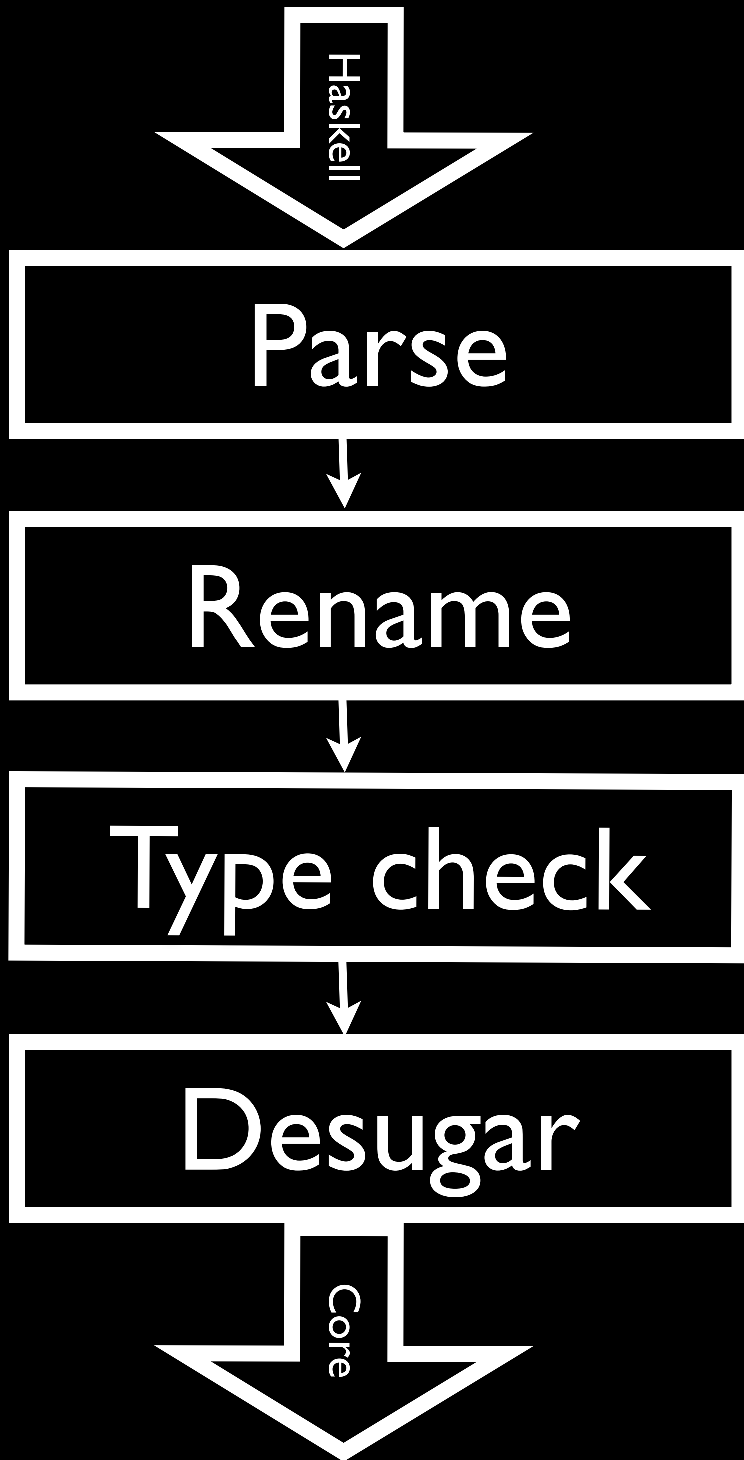
Solving the expression problem in Haskell: Part I

Sean Seefried

Front-end Plugins



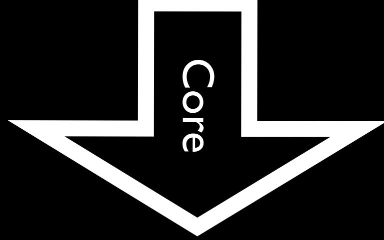
Haskell
Parse
Rename
Type check
Desugar
Core

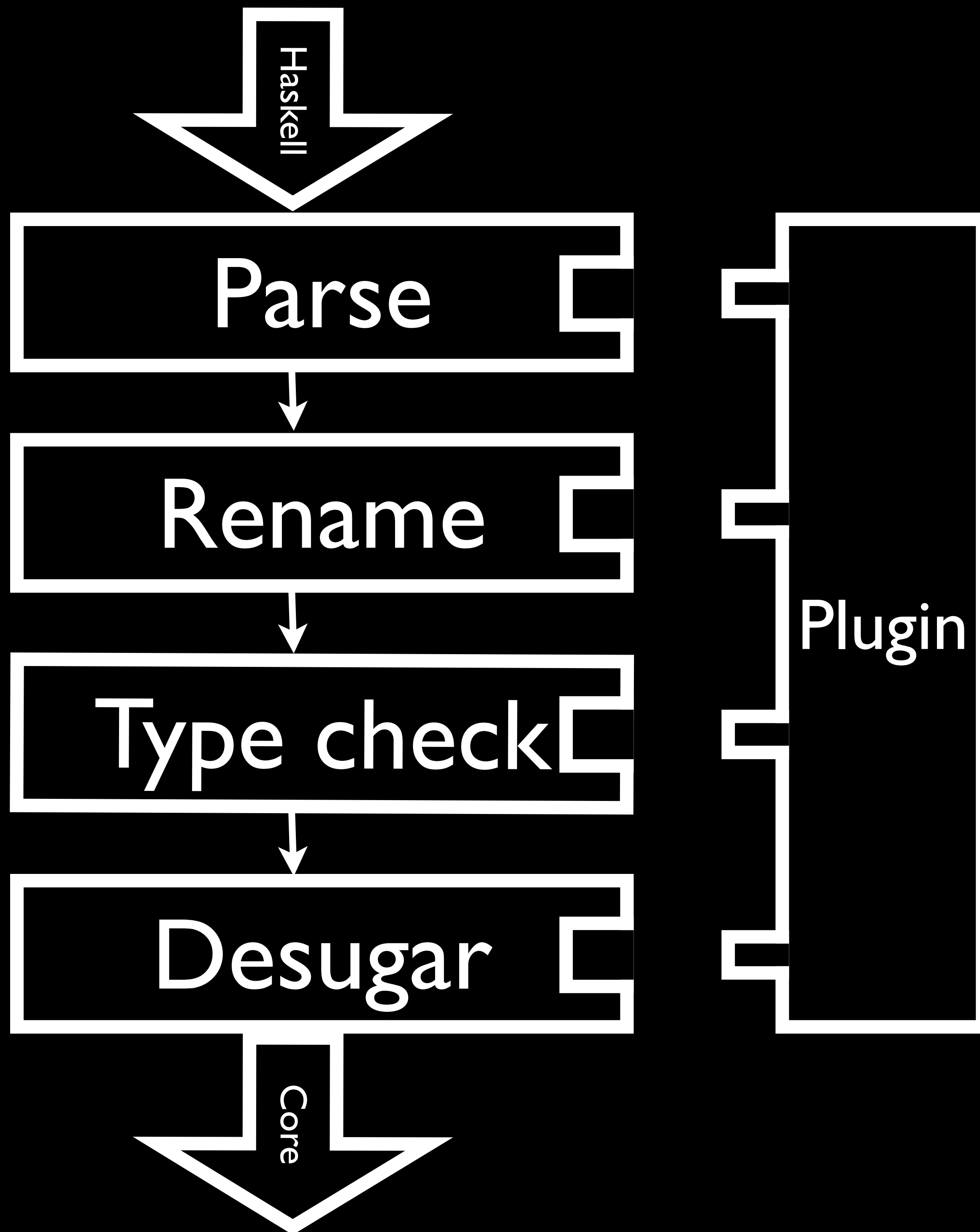
Parse

Rename

Type check

Desugar




Haskell
Parse
Rename
Type check
Desugar
Core
Plugin

Extending data types

```
data Exp = VarE String
         | LamE String Exp
         | AppE Exp Exp
```

```
eval :: Env -> Exp -> Exp
eval env (VarE v) = ...
eval env (LamE v body) = ...
eval env (AppE e e’) = ...
```

```
data Exp = VarE String
         | LamE String Exp
         | AppE Exp Exp
         | LetE Name Exp Exp
```

```
eval :: Env -> Exp -> Exp
eval env (VarE v) = ...
eval env (LamE v body) = ...
eval env (AppE e e’) = ...
eval env (LetE v exp body) = ...
```

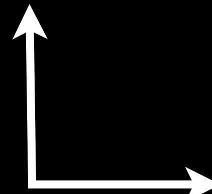
```
eval :: Env -> Exp -> Exp
eval env (VarE v) = ...
eval env (LamE v body) = ...
eval env (AppE e e') = ...
eval env (LetE v exp body) = ...
```

Plugins *need* EDTs

Expression Problem

The expression problem

Extensibility in both dimensions



Strong static type safety

No modification or duplication

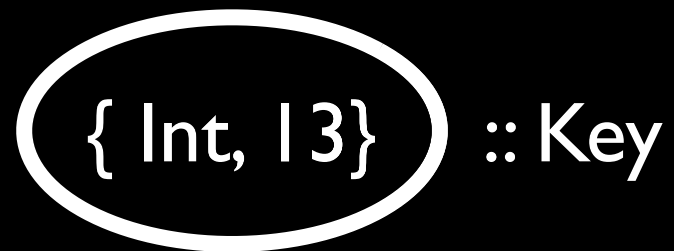
Separate compilation

Open vs. Closed

Type classes are open

Existential types

```
data Key = forall a. MkKey a
```



:: Key

```
data Key = forall a. MkKey a
```

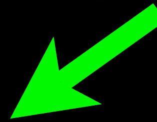
:: Key

Existential types
+
Type classes

```
class Key_ a where
   getKey :: a -> Int

instance Key_ Int where getKey = id

instance Key_ Bool where
  getKey x = if x then 1 else 0

instance Key_ [a] where
  getKey = length
```

```
data Key = forall a.Key_ a => MkKey a
```

Unwrapping instance



```
instance Key_ Keywhere
  getKey (MkKey a) = getKey a
```

```
data Key= forall a. Key_ a => MkKey a
```

getKey :: Int -> Int
getKey = id

{ Int, 13}

:: Key

```
data Key= forall a. Key_ a => MkKey a
```

Can we solve the expression problem yet?

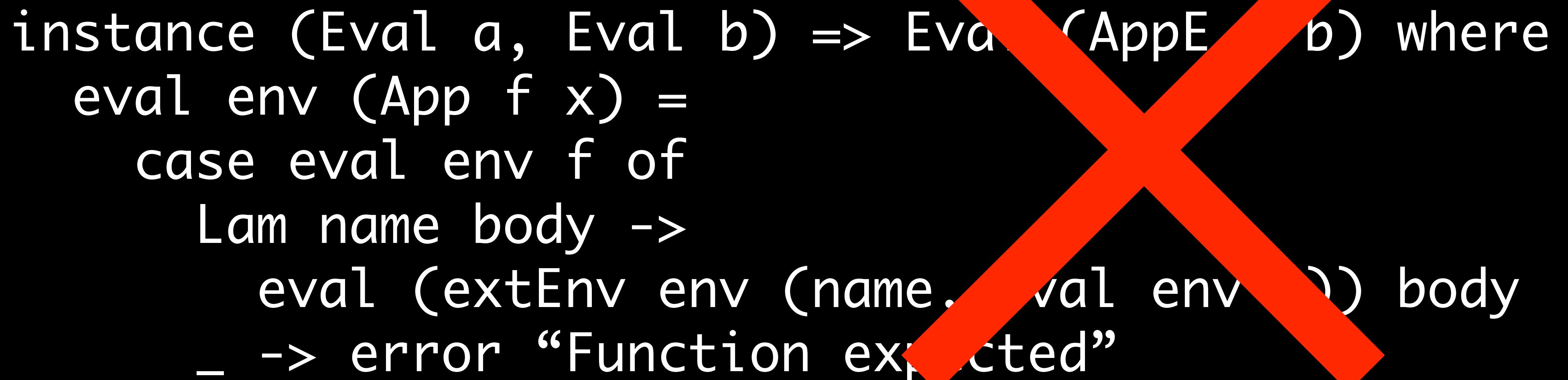
```
class Exp_ a
data Exp = forall a. Exp_ a => MkExp a

data Var      = Var String
data Lam a    = Lam String a
data App a b  = App a b

instance Exp_ Var
instance (Exp_ a) => Exp_ (Lam a)
instance (Exp_ a, Exp_ b) => Exp_ (App a b)

instance Exp_ Exp
```

Extending functionality

```
data EvalExp = forall a. Eval a => MkEvalExp a

type Env = [(String, EvalExp)]

class Exp_ a => Eval a where
  eval :: Env -> a -> EvalExp
  eval _ exp = MkEvalExp exp
```

```
instance Eval Var where
  eval env (Var name) = lookupEnv env name

instance Eval a => Eval (Lam a)
  <default method is good enough>

instance (Eval a, Eval b) => Eval (AppE a b) where
  eval env (App f x) =
    case eval env f of
      Lam name body ->
        eval (extEnv env (name, eval env x)) body
      _ -> error “Function expected”
```

```
instance Eval Var where
  eval env (Var name) = lookupEnv env name

instance Eval a => Eval (Lam a)
  <default method is good enough>
```



Can’t pattern match on existentially wrapped things

```
class Exp_ a => Eval a where
  eval :: Env -> a -> EvalExp
  eval _ exp = MkEvalExp exp
```

```
class Exp_ a => Eval a where
  eval :: Env -> a -> EvalExp
  eval _ exp = MkEvalExp exp
  apply :: Eval x => Env -> x -> a -> EvalExp
  apply = error "Function expected"
```

```
class Exp a => Interpreter a where
  eval :: Env -> a -> IEXP
  eval _ exp = MkIExp exp
  apply :: Eval x => Env -> x -> a -> EvalExp
  apply = error “Function expected”

instance Eval a => Eval (Lam a) where
  apply env x (Lam name body) =
    eval (extEnv (name, eval env x)) body

instance (Eval a, Eval b) => Eval (App a b) where
  eval env (MkEvalExp exp) =
    apply env x (eval env f)
```

Pattern matches become new class methods

Oh, and don't forget to unwrap!

```
instance Exp_ EvalExp

instance Eval EvalExp where
  eval env (MkEvalExp exp) = eval env exp
  apply env x (MkIExp exp) = apply env x exp
```

Demo

```
> let term = Lam “x” (Var “x”)
> eval [] (App term term)
Lam “x” (Var “x”)
```

```
> let term = Lam “x” (Var “x”)
> eval [] (App term term)
Lam “x” (Var “x”)

> :t eval [] (App term term)
eval [] (App term term) :: EvalExp
```

```
> let term = Lam “x” (Var “x”)
> eval [] (App term term)
Lam “x” (Var “x”)

> :t eval [] (App term term)
eval [] (App term term) :: EvalExp

> let term’ = eval [] (App term term)
> eval (App term’ term)
Lam “x” (Var “x”)
```

Adding a new variant

```
data Let a b = Let String a b

instance (Exp_ a, Exp_ b) => Exp_ (Let a b)

instance (Eval a, Eval b) => Eval (Let a b) where
  eval env (Let name exp body) =
    eval env (App (Lam name body) exp)
```

```
> let term = Lam “y” (Var “y”)
> let body = App (Var “x”) (Var “x)
> eval (Let “x” term body)
Lam “y” (Var “y”)
```

Adding more functionality

```
class Eval a => IsVar a where
  isVar :: a -> Bool
  isVar _ = False

instance IsVar Var where
  isVar _ = True

data IsVarExp = forall a. IsVar a => MkIsVarExp a
```

Unwrapping instances

```
instance Exp IsVarExp

instance Interpreter IsVarExp where
  eval env (MkIsVarExp exp) = eval env exp

instance IsVarExp IsVarExp where
  isVar (MkIsVarExp e) = isVar e
```

There’s a problem!

```
> let term = Lam “x” (Var “x”)
> eval [] term
Lam “x” (Var “x”)

> isVar (eval [] term)
```

```
> let term = Lam “x” (Var “x”)
> eval [] term
Lam “x” (Var “x”)

> isVar (eval [] term)

 No instance for (IsVar IsVarExp)
    arising from use of `isVar' at <interactive>:1:0-4
  Probable fix: add an instance declaration for (IsVar IsVarExp)
  In the definition of `it': it = isVar (eval [] term)
```

One existential type to rule them all

Wouldn't it be cool if we could abstract on classes?

```
data Exp cxt = forall a. cxt a => MkExp a
```

Questions?