eDSLs and interpreters

Motivation

- ADTs are great for representing expressions and programs
  (we'll have a look at some tools and alternatives)
- embedded languages carry minimal overhead (thanks to aggressive optimization)
- effect systems make implementation of interpreters quite straightforward

Extensible and analyzable data structures

```
data Expr
    = Var String
    | Lam String Expr
    | App Expr Expr
    | Plus Expr Expr
    | Negate Expr
    | ...
```

- Is string a good type for variables names?
- Are my functions going to work if I add some new language construction?
- Is the expression valid (w.r.t. some semantics)?
- Do I have to use recursion in all processing functions?
- ...

Common tool: parametrize the binder type

```
data Expr a
  = Var a
  | Lam a (Expr a)
  | App (Expr a) (Expr a)
  | Plus (Expr a) (Expr a)
  | Negate (Expr a)
  | ...
```

- makes sense as a Functor
- instead of Strings we can have integers and references (for performance)

Higher Order Abstract Syntax

```
data Expr a where
  Val :: a → Expr a
  Lam :: (Expr a → Expr b) → Expr (a → b)
  App :: Expr (a → b) → Expr a → Expr b

id' :: Expr (a → a)
id' = Lam (λx → x)

const' :: Expr (a → b → a)
const' = Lam (λx → Lam (λy → x))

eval :: Expr a → a
eval (Val v) = v
eval (Lam f) = λx → eval (f (Val x))
eval (App f p) = (eval f) (eval p)
```

- the expressions are certainly well-typed!
- mild inconveniences: parsers have to know the types in advance, prettyprinting is hard, ...

Problem: A structure in ADT syntax does not offer any type-related guarantees (but can be deconstructed manually). HOAS can't be pattern-matched (but the types are checked by Haskell).

Final interpreters

```
class Expr repr where
  val :: Int → repr
  add :: repr → repr → repr
  mul :: repr → repr → repr

instance Expr Int where
  val n = n
  add a b = a + b
  mul a b = a * b

instance Expr String where
  val n = show n
  add a b = "(" ++ a ++ "+" ++ b ++ ")"
  mul a b = "(" ++ a ++ "*" ++ b ++ ")"
```

```
eval :: Int → Int
eval = id

render :: String → String
render = id

expr :: Expr r ⇒ r
expr = add (val 1) (mul (val 2) (val 3))

eval expr ≡ 7
render expr ≡ "(1+(2*3))"
```

Final interpreters — benefits

Extends horizontally:

```
instance Expr (Writer [Asm]) where ...

compile :: Writer [Asm] → Writer [Asm]
compile = id
```

Extends vertically:

```
class Expr repr ⇒ Lang repr
  var :: String → repr
  assign :: String → repr → repr
  semicolon :: repr → repr → repr

instance Lang (Writer [Asm]) where
  ...
  semicolon a b = a ≫ b
```

What does the final encoding look like in the memory?

What does the final encoding look like in the memory?

AST made of functions closures connected in STG.

Tagless final

```
class Expr rep where
  lam :: (rep a → rep b) → rep (a → b)
  app :: rep (a → b) → (rep a → rep b)
  val :: a → rep a

newtype Interpret a = R {reify :: a}

instance Expr Interpret where
  lam f = R $ reify . f . R
  app f a = R $ reify f $ reify a
  val = R

example :: Expr rep ⇒ rep Int
example = app (lam (λx → x)) (val 3)

eval :: Interpret a → a
eval e = reify e
```

- tagged final interpreters have to store the “type” information themselves
- “types” in the tagless version are managed by the compiler, and there’s no need to store or match them

Interpreter nomenclature

Tagged holds the AST and possibly its “types”

Tagless “types” are embedded into the implementation language and held by compiler (solves static safety)

Initial type holds the structure, interpretation consumes the structure

Final interpretation is injected into structure creation (solves extensibility)

	Tagged	Tagless
Initial	ADT (**data** Expr)	HOAS (**data** Expr *t*)
Final	generators of *repr*	generators of *repr t*

What follows is an assorted collection of useful solutions to various annoyances of interpreter implementation.

Code annotation (e.g., locations)

```
type Loc = Int
data Expr
    = Val Loc Int
    | Plus Loc Expr Expr
```

Code annotation (e.g., locations)

```
type Loc = Int
data Expr
    = Val Loc Int
    | Plus Loc Expr Expr
```

A slightly more systematic solution:

```
data L a = L Int a
type Expr a = L (Ex a)
data Ex a
    = Val Int
    | Plus Expr Expr
```

Code annotation with Higher Kinded Data

Systematically via HKD (Data.Fix):

```
data Expr f
  = Val Int
  | Plus (f (Expr f)) (f (Expr f))

data Loc a = Loc Int a

type LocExpr = Expr Loc

plus (Loc 0 (Val 1234)) (Loc 1 (Val 2345) :: LocExpr
```

Benefits:

- Expr Identity, Expr WithInferredType
- Expr (Const TreeIntRef) (converts recursion into explicit one)
- Expr IO
- Cool libraries available (`barbies`)

Fixing the manual recursion in HKDs

```
import Data.Fix

data Expr a = Val Int | Plus a a
data Loc f a = Loc Int (f a)

loc l = Fix . Loc l

loc 0 (Val 3) :: Fix (Loc Expr)
```

Let's just have functors, shall we?

```
import Data.Fix
import Data.Functor.Compose

data Expr a = Val Int | Plus a a
data Loc a = Loc Int a

loc' n = Fix . Compose . Loc n

loc' 0 (Val 3) :: Fix (Compose Loc Expr)
```

Side note: monad transformers are technically HKDs


Vorsicht
Funktor
2 m Abstand halten

Monadic DSL without monads

You can make a monad out of any functor “for free” (Control.Monad.Free):

```
data Free f a
  = Pure a
  | Free (f (Free f a))

instance Functor f ⇒ Monad (Free f)
  where
    return        = Pure
    Pure a >>= f  = f a
    Free m >>= f = Free (fmap (>>=f) m)
```

Monadic DSL without monads

You can make a monad out of any functor “for free” (Control.Monad.Free):

```
data Free f a
  = Pure a
  | Free (f (Free f a))

instance Functor f ⇒ Monad (Free f)
  where
    return        = Pure
    Pure a >>= f  = f a
    Free m >>= f  = Free (fmap (>>=f) m)
```

- Pure values behave “normally”
- Each use of Free creates a layer of functor (which can be interpreted later)

```
Just 3 >> Nothing
    ⇝ Nothing

Free (Just $ pure 3) >> Free Nothing
    ⇝ Free (Just (Free Nothing))

Pure 123 >>= λx → return x
    ⇝ Free (Just (Pure 123))

Free (Just (Pure 1)) >> Free (Just (Pure 3))
    ⇝ Free (Just (Free (Just (Pure 3))))

sum <$> traverse (λx → Free (x, pure x)) [1..5]
    ⇝ Free (1, Free (2, Free (3, Free (4, Free (5,
          Pure 15)))))
```

Use of free monads

```
data ConsoleCmd a
    = WriteLn String a
    | ReadLn (String → a)
  deriving Functor

type ConsoleM = Free ConsoleCmd

liftF :: Functor f ⇒ f a → Free f a
liftF = Free ∘ fmap Pure

writeLn s = liftF (WriteLn s ())
readLn = liftF (ReadLn id)

program =
  x ← readLn
  y ← readLn
  writeLn (show $ read x + read y)
```

Use of free monads

```
data ConsoleCmd a
  = WriteLn String a
  | ReadLn (String → a)
  deriving Functor

type ConsoleM = Free ConsoleCmd

liftF :: Functor f ⇒ f a → Free f a
liftF = Free . fmap Pure

writeLn s = liftF (WriteLn s ())
readLn = liftF (ReadLn id)

program =
  x ← readLn
  y ← readLn
  writeLn (show $ read x + read y)
```

```
consInIO :: ConsoleCmd a → IO a
consInIO (WriteLn s cont) = do
  putStrLn s
  pure cont
consInIO (ReadLn callback) = do
  s ← getLine
  pure (callback s)

-- Control.Monad.Free
foldFree :: Monad m
         ⇒ (∀x. f x → m x)
         → Free f a → m a

main = foldFree consInIO program
```

Interpret can be switched easily, e.g. *consInMock*, *consInConduit*, ...

Selective

- Applicative functors can describe quite complex processes, but they can not decide about the “shape” of the computation (and thus about side effects) based on intermediate results.
- Monads can do this with $\gg=$, but the code has to be executed (interpreted) to allow analysis.

Selective

- Applicative functors can describe quite complex processes, but they can not decide about the “shape” of the computation (and thus about side effects) based on intermediate results.
- Monads can do this with $\gg=$, but the code has to be executed (interpreted) to allow analysis.

$$\textbf{class}\ \text{Applicative}\ f \Rightarrow \text{Selective}\ f\ \textbf{where}$$
$$\quad \mathit{select} :: f\ (\text{Either}\ a\ b) \to f\ (a \to b) \to f\ b$$
$$\mathit{ifS} :: \text{Selective}\ f \Rightarrow f\ \text{Bool} \to f\ a \to f\ a \to f\ a$$

Benefit: with the right functor it is possible to collect information about all possible code branches (e.g., all kinds of error states).
(similar: `optparse-applicative`)

Selective

- Applicative functors can describe quite complex processes, but they can not decide about the “shape” of the computation (and thus about side effects) based on intermediate results.
- Monads can do this with $\gg\!=$, but the code has to be executed (interpreted) to allow analysis.

class Applicative $f \Rightarrow$ Selective f **where**
  $select :: f\ (\text{Either}\ a\ b) \to f\ (a \to b) \to f\ b$

$ifS :: \text{Selective}\ f \Rightarrow f\ \text{Bool} \to f\ a \to f\ a \to f\ a$

Benefit: with the right functor it is possible to collect information about all possible code branches (e.g., all kinds of error states).
(similar: `optparse-applicative`)

```
data Shape = Circle Int | Rectangle Int Int

shape :: Selective f ⇒
         f Bool → f Int → f Int → f Int → f Shape
shape s r w h = ifS s (Circle ⟨$⟩ r) (Rectangle ⟨$⟩ w ⟨*⟩ h)

shape (Success True) (Success 10)
                     (Failure ["no width"])
                     (Failure ["no height"])
  ↝ Success (Circle 10)

shape (Success False) (Failure ["no radius"])
                      (Failure ["no width"])
                      (Failure ["no height"])
  ↝ Failure ["no width", "no height"]
```

✎Mokhov, A., Lukyanov, G., Marlow, S., & Dimino, J. (2019). *Selective applicative functors.* Proceedings of the ACM on Programming Languages, 3(ICFP), 1-29.

Take-home message

Functor $<$ Applicative $\leqslant$ Selective $<$ Monad

Extensibility

Programs that work with complex data structures typically extend as follows:

Vertically by adding data types
(OOP: by adding more classes, ADT: by adding more alternatives to a sum type, ...)

Horizontally by adding functionality
(OOP: by extending the interfaces, ADT: by adding functions/instances)

Problem: You need $O(h \cdot v)$ of code; even trivial extensions require implementation of $O(h + v)$ functions, typically 'boilerplate'.

Goal: We only want to implement the interesting stuff.

Can we traverse (and modify) structures generically?

(First we have to materialize some type safety.)

class Typeable a – methods omitted for brevity

$cast$ :: (Typeable a, Typeable b) $\Rightarrow$ $a \rightarrow$ Maybe b

($cast$ `'a'`) :: Maybe Bool
$\rightsquigarrow$ Nothing
($cast$ True) :: Maybe Bool
$\rightsquigarrow$ Just True

✎Lämmel, R., & Jones, S. P. (2003). *Scrap your boilerplate: a practical design pattern for generic programming.* ACM SIGPLAN Notices, 38(3), 26-37.

Generic transformations don't do anything on unknown types:

```
mkT :: (Typeable a, Typeable b)
    ⇒ (a → a) → b → b
mkT f = case cast f of
  Just g → g
  Nothing → id

mkT (¬) True
    ⇝ False

mkT (¬) 'a'
    ⇝ 'a'
```

With some knowledge of the 'data shape', we can run generic transformations on whole data types:

```
class Typeable a ⇒ Term a where
  gmapT :: (∀b.Term b ⇒ b → b) → a → a

instance Term Either a b where
  gmapT f (Left l) = Left (f l)
  gmapT f (Right r) = Right (f r)

instance Term (a, b) where
  gmapT f (l, r) = (f l, f r)
```

With some knowledge of the 'data shape', we can run generic transformations on whole data types:

```
class Typeable a ⇒ Term a where
  gmapT :: (∀b.Term b ⇒ b → b) → a → a

instance Term Either a b where
  gmapT f (Left l) = Left (f l)
  gmapT f (Right r) = Right (f r)

instance Term (a, b) where
  gmapT f (l, r) = (f l, f r)
```

With some knowledge of the 'data shape', we can run generic transformations on whole data types:

```
class Typeable a ⇒ Term a where
  gmapT :: (∀b.Term b ⇒ b → b) → a → a

instance Term Either a b where
  gmapT f (Left l) = Left (f l)
  gmapT f (Right r) = Right (f r)

instance Term (a, b) where
  gmapT f (l, r) = (f l, f r)


gmapT (mkT (¬)) (True, 'a')
    ⇝ (False, 'a')
```

Recursively:

$$
\begin{aligned}
&\mathit{everywhere} :: \mathrm{Term}\ a \\
&\qquad \Rightarrow (\forall b.\mathrm{Term}\ b \Rightarrow b \to b) \\
&\qquad \to a \to a \\
&\mathit{everywhere}\ f\ x = f\ (\mathit{gmapT}\ (\mathit{everywhere}\ f)\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{everywhere}\ (\mathit{mkT}\ (+1))\ (\mathrm{True}, \texttt{'a'}, (1, \mathrm{False}, [\mathrm{True}, \mathrm{False}])) \\
&\quad \rightsquigarrow (\mathrm{True}, \texttt{'a'}, (2, \mathrm{False}, [\mathrm{True}, \mathrm{False}]))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{everywhere}\ (\mathit{mkT}\ (\neg))\ (\mathrm{True}, (1, \mathrm{False}, [\mathrm{True}, \mathrm{False}])) \\
&\quad \rightsquigarrow (\mathrm{False}, (1, \mathrm{True}, [\mathrm{False}, \mathrm{True}]))
\end{aligned}
$$

```
class Typeable a where
  typeRep# ::TypeRep a

class Typeable a ⇒ Data a where
  gfoldl :: (∀d b.Data d ⇒ c (d → b) → d → c b) → (∀g.g → c g) → a → c a
  gunfold, toConstr, gmapT, gmapQ, gmapM, ...
```

Both classes can be derived mechanically! Data additionally manages monads and type casts:

```
mkT :: (Typeable a, Typeable b) ⇒ (b → b) → a → a
mkQ :: (Typeable a, Typeable b) ⇒ r → (b → r) → a → r
mkM :: (Monad m, Typeable a, Typeable b) ⇒ (b → m b) → a → m a
mkR :: (MonadPlus m, Typeable a, Typeable b) ⇒ m b → m a
```

Generics and others

Traditional languages

- `Typeable` roughly corresponds to RTTI, `Data` is very roughly a "Serializable" trait
- 'visitor pattern' is a similar idea, but does not solve the $\mathcal{O}(h \cdot v)$ issue and usually can not be derived automatically.

GHC.Generics Auto-generated typeclass `Generic` gives comprehensive information about hte GHC type, including e.g. constructor names.

```
class Generic a where
  type Rep a :: Type → Type
  from :: a → (Rep a) x
  to   :: (Rep a) x → a
```

Exemplary use: Default typeclass methods in Data.Aeson.

Uniplate/Biplate Slightly older (and simpler) representation of types based on *from&to*; derivable from `Data`.

Use of generics

- auto-derive FromJSON and ToJSON instances for types (similarly in many other encoding-ish packages)
- avoid TemplateHaskell
- auto-generate APIs for types
- auto-generate lenses
- differentiate types (auto-zipper construction)
- modify types (Data.Surgery)

```
ghci> from (Just 3)
M1 {unM1 = R1 (M1 {unM1 = M1 {unM1 = K1 {unK1 = 3}}})}

ghci> :k! Rep (Maybe Int)
Rep (Maybe Int) :: * -> *
= D1
    ('MetaData "Maybe" "GHC.Maybe" "base" 'False)
    (C1 ('MetaCons "Nothing" 'PrefixI 'False) U1
     :+: C1
           ('MetaCons "Just" 'PrefixI 'False)
           (S1
              ('MetaSel 'Nothing
                 'NoSourceUnpackedness 'NoSourceStrictness 'DecidedLazy)
              (Rec0 Int)))
```

Bonus: Optics without boilerplate

```
import Control.Lens.Plated

incrVals = transformOf _Val (+1) :: Expr → Expr
incrVals (Plus (Val 1) (Mul (Val 2) (Val 3)))
    ⇝ (Plus (Val 2) (Mul (Val 3) (Val 4)))

(Plus (Val 1) (Var "x")) ^.. cosmos . _Var
    ⇝ ["x"]

import Data.Aeson.Lens
"[1, \"x\", {\"y\": 123}]" ^.. cosmos . _Number
    ⇝ [1,123]
```

Final note: interpreters are insanely powerful

Partial evaluator is a program that converts a source code that contains a static (known) and dynamic (unknown) part to source code that contains no statically derivable computation.

$$
\begin{aligned}
&\mathit{mix} :: \text{Program} \\
&\quad \rightarrow \text{StaticInput} \\
&\quad \rightarrow \text{Program}
\end{aligned}
$$

Similar: $s_{m,n}$ theorem in enumerability, *peval*, "specialization"

Futamura projections (1970):

0. Interpretation of source code:
   interp source input = mix source input
   (returns a program that prints out the result without any other computation)
1. Compilation of a program:
   mix interp source
2. Creating a compiler program:
   mix mix interp
3. Creating a universal compiler of interpreters to compilators:
   mix mix mix

The End!

...but what comes next?

What next?

- Interpreters and compilers! Haskell is great for these!
- GADTs, existential types (`GenericList`)
- DataKinds (API `"items"`), dependent types (`Array 5 Int`)
- recursion schemes
- effect systems (`freer`, `polysemy`, `fused-effects`, `eff`)
- profunctors and profunctor optic
- linear types