Framing the Discussion

Anthony Cowley

Tagged Initial

```haskell
data LangI a where
  LiftI :: a -> LangI a
  AddI :: LangI Int -> LangI Int -> LangI Int
  LamI :: (LangI a -> LangI b) -> LangI (a -> b)
  AppI :: LangI (a -> b) -> LangI a -> LangI b
```

Interpreter

```
interp :: LangI a -> a
interp (LiftI x) = x
interp (AddI x y) = interp x + interp y
interp (LamI f) = interp . f . LiftI
interp (AppI f x) = interp f (interp x)

instance Num a => Num (LangI a) where
  fromInteger = LiftI . fromInteger
```

Example

```
testI1 :: Int
testI1 = interp (AddI 2 3)
```

```
> testI1
5
```

```
testI2 :: Int
testI2 = interp (AppI (LamI $ \x -> AddI 2 x) 3)
```

```
> testI2
5
```

```
testI2' :: Int
testI2' = interp (AppI (LamI $ AddI 2) 3)
```

Tagless Final

```haskell
class ArithF e where
  lit :: Int -> e Int
  add :: e Int -> e Int -> e Int

class AbsF e where
  lam :: (e a -> e b) -> e (a -> b)
  app :: e (a -> b) -> e a -> e b

class FancyF e where
  fancyOp :: e Int -> e Int
```

Our first compiler backend

```
instance ArithF Identity where
  lit = pure
  add = liftA2 (+)

instance AbsF Identity where
  lam f = Identity (runIdentity . f . Identity)
  app = (<*>)

instance FancyF Identity where
  fancyOp = fmap (+42)
```

Compositional Language

```haskell
type MyLang e = (ArithF e, AbsF e)

testSum :: MyLang e => e Int
testSum = add (lit 2) (lit 3)
```

Compositional Language

```
type MyLang e = (ArithF e, AbsF e)

testSum :: MyLang e => e Int
testSum = add (lit 2) (lit 3)
```

```
> runIdentity testSum
5
```

Code Generation

```
newtype Code a = Code { getCode :: Int -> String }

instance ArithF Code where
  lit = Code . const . show
  add (Code x) (Code y) = Code $ \n ->
    "(" ++ x n ++ " + " ++ y n ++ ")"

instance AbsF Code where
  lam f = Code $ \n ->
          let  x = "x_"++show n
               Code body = f (Code $ const x)
               subst = body (n+1)
          in   concat ["(\\",x," -> ",subst,")"]
  app (Code f) (Code x) =
    Code $ \n -> concat ["(",f n," ",x n,")"]
```

Hardware SDK

```
instance FancyF Code where
  fancyOp (Code x) =
    Code $ \n -> "(hardwareOperation "++x n++")"
```

Code Generation Test

```
> getCode testSum 0
"(2 + 3)"
```

Code Generation Test

```
> getCode testSum 0
"(2 + 3)"
```

```haskell
testLam :: MyLang e => e (Int -> Int)
testLam = lam $ \x -> add x x
```

```
> putStrLn $ getCode testLam 1
(\x_1 -> (x_1 + x_1))
```

Indexed Initial Encoding

```
data LangIG = ArithIG | FancyIG
```

Singletons

```
data family LangSing :: k -> *

data instance LangSing (a::LangIG) where
  SArithIG :: LangSing ArithIG
  SFancyIG :: LangSing FancyIG

class ISing (a :: k) where sing :: LangSing a
instance ISing ArithIG where sing = SArithIG
instance ISing FancyIG where sing = SFancyIG
```

Indexed Initial Encoding

Modularly Tagged

```
data family Repr :: k -> [k] -> (* -> *) -> * -> *

data TermIG :: [LangIG] -> (* -> *) -> * -> * where
  TermIG :: (El lang langs)
         => LangSing lang
         -> Repr lang langs e a
         -> TermIG langs e a
```

Defining a tagged sub-language

A Family of Tags

```
data instance Repr ArithIG langs e a where
  LitIG :: Int -> Repr ArithIG langs e Int
  AddIG :: TermIG langs e Int
        -> TermIG langs e Int
        -> Repr ArithIG langs e Int

data instance Repr FancyIG langs e a where
  FancyOpIG :: TermIG langs e Int
            -> Repr FancyIG langs e Int
```

Helper

```
termIG :: (El lang langs, ISing lang)
       => Repr lang langs e a -> TermIG langs e a
termIG = TermIG sing
```

Indexed Initial Evaluation

```
type MyLangIG = [ArithIG, FancyIG]
```

Indexed Initial Evaluation

```haskell
type MyLangIG = [ArithIG, FancyIG]
```

```haskell
evalIG :: TermIG MyLangIG e a -> a
evalIG (TermIG SArithIG (LitIG x)) = x
evalIG (TermIG SArithIG (AddIG x y)) =
  evalIG x + evalIG y
evalIG (TermIG SFancyIG (FancyOpIG x)) =
  evalIG x + 42
```

Indexed Initial testSum

```
testSumIG :: TermIG MyLangIG e Int
testSumIG = termIG (AddIG (termIG (LitIG 2))
                          (termIG (LitIG 3)))
```

```
> evalIG testSumIG
5
```

Modular Evaluation

```haskell
data LangME = ArithME | FancyME

class EvalME (lang :: LangME) where
  evalME :: (forall a. TermME langs Identity a -> a)
         -> Repr lang langs Identity r -> r

data TermME :: [LangME] -> (* -> *) -> * -> * where
  TermME :: (El lang langs, EvalME lang)
         => LangSing lang
         -> Repr lang langs e a
         -> TermME langs e a

termME :: (El lang langs, ISing lang, EvalME lang)
       => Repr lang langs e a -> TermME langs e a
termME = TermME sing
```

Modular Evaluation Implementation (Boring)

```
data instance Repr ArithME langs e a where
  LitME :: Int -> Repr ArithME langs e Int
  AddME :: TermME langs e Int
        -> TermME langs e Int
        -> Repr ArithME langs e Int

data instance Repr FancyME langs e a where
  FancyOpME :: TermME langs e Int
            -> Repr FancyME langs e Int
```

Modular Implementation

```
instance EvalME ArithME where
  evalME _ (LitME x) = x
  evalME k (AddME x y) = k x + k y

instance EvalME FancyME where
  evalME k (FancyOpME x) = k x + 42
```

Modular Evaluation

```
type MyLangME = [ArithME, FancyME]

runEvalME :: (forall e. TermME MyLangME e a) -> a
runEvalME t = go t
  where go :: TermME MyLangME Identity a -> a
        go (TermME _ x) = evalME go x

testSumME ::TermME MyLangME e Int
testSumME = termME (AddME (termME (LitME 2))
                          (termME (LitME 3)))
```

```
> runEvalME testSumME
5
```

Finally Initial

evalME? I'll eval you!

```
data LangFI = ArithFI | FancyFI

type family Finally (l :: k) (e :: * -> *) :: Constraint

class EvalFI (lang :: LangFI) where
  evalFI :: Finally lang e
         => (forall a. TermFI langs e a -> a)
         -> Repr lang langs e r -> e r
```

Finally Terms (Boring)

```
data TermFI :: [LangFI] -> (* -> *) -> * -> * where
  TermFI :: (El lang langs, EvalFI lang, Finally lang e)
         => LangSing lang
         -> Repr lang langs e a
         -> TermFI langs e a
```

Finally Sub-languages (Boring)

```
data instance Repr ArithFI langs e a where
  LitFI :: Int -> Repr ArithFI langs e Int
  AddFI :: TermFI langs e Int
        -> TermFI langs e Int
        -> Repr ArithFI langs e Int

data instance Repr FancyFI langs e a where
  FancyOpFI :: TermFI langs e Int
            -> Repr FancyFI langs e Int
```

Final Evaluation

Oh right, we defined these a long time ago!

```
type instance Finally ArithFI e = ArithF e

instance EvalFI ArithFI where
  evalFI _ (LitFI x) = lit x
  evalFI k (AddFI x y) = add (lit (k x)) (lit (k y))

type instance Finally FancyFI e =
  (ArithF e, FancyF e)

instance EvalFI FancyFI where
  evalFI k (FancyOpFI x) = fancyOp (lit (k x))
```

Interpreting Finally Initial

```
type MyLangFI = [ ArithFI, FancyFI ]

runEvalFI :: (forall e.  AllFinal MyLangFI e
              => TermFI MyLangFI e a)
          -> a
runEvalFI t = go t
  where go :: TermFI MyLangFI Identity b -> b
        go (TermFI _ x) = runIdentity (evalFI go x)
```

Finally Initial testSum

```
testSumFI :: (El ArithFI langs, ArithF e)
          => TermFI langs e Int
testSumFI = termFI (AddFI (termFI (LitFI 2))
                          (termFI (LitFI 3)))
```

```
> runEvalFI testSumFI :: Int
5
```

Partially Tagless

```
data LangPF = ArithPF | FancyPF | AbsPF

class EvalPF (lang :: LangPF) where
  evalPF :: (El lang langs, Finally lang e)
         => (forall a. TermPF langs e a -> e a)
         -> Repr lang langs e r -> e r
```

Potentially Partial Evaluation

```
class PEval (lang :: LangPF) where
  pevalPF :: (El lang langs,
              Finally lang (TermPF langs e))
          => (forall a. TermPF langs (TermPF langs e) a
              -> TermPF langs e a)
          -> Repr lang langs (TermPF langs e) r
          -> TermPF langs e r
  default pevalPF
    :: (El lang langs, EvalPF lang,
        Finally lang (TermPF langs e))
    => (forall a. TermPF langs (TermPF langs e) a
        -> TermPF langs e a)
    -> Repr lang langs (TermPF langs e) r
    -> TermPF langs e r
  pevalPF = evalPF
```

Terms Look the Same

```
data TermPF :: [LangPF] -> (* -> *) -> * -> * where
  TermPF :: (El lang langs, EvalPF lang, PEval lang,
              Finally lang e )
          => LangSing lang
          -> Repr lang langs e a
          -> TermPF langs e a
```

Term Data Types Look the Same

```
data instance Repr ArithPF langs e a where
  LitPF :: Int -> Repr ArithPF langs e Int
  AddPF :: TermPF langs e Int
        -> TermPF langs e Int
        -> Repr ArithPF langs e Int
```

Terms as Finally Tagless Backends

```
instance (El ArithPF langs, Finally ArithPF e)
  => ArithF (TermPF langs e) where
  lit = termPF . LitPF
  add x y = termPF (AddPF x y)
```

Evaluation... still Final

```
type instance Finally ArithPF e = ArithF e

instance EvalPF ArithPF where
  evalPF _ (LitPF x) = lit x
  evalPF k (AddPF x y) = add (k x) (k y)
```

Partial Evaluation

Not Boring

```
pattern AsLit x <- TermPF SArithPF (LitPF x)

litPF :: (ArithF e, El ArithPF langs)
      => Int -> TermPF langs e Int
litPF = termPF . LitPF

instance PEval ArithPF where
  pevalPF _ (LitPF x) = lit x
  pevalPF k (AddPF x y) =
    case (k x, k y) of
      (AsLit x', AsLit y') -> litPF $ x' + y'
      (x', y') -> add x' y'
```

Bringing Lambda Back

```
data instance Repr AbsPF langs e a where
  LamPF :: (TermPF langs e a -> TermPF langs e b)
        -> Repr AbsPF langs e (a -> b)
  AppPF :: TermPF langs e (a -> b)
        -> TermPF langs e a
        -> Repr AbsPF langs e b
  VarPF :: e a -> Repr AbsPF langs e a

type instance Finally AbsPF e = AbsF e
```

Evaluation

```
instance (El AbsPF langs, AbsF e)
  => AbsF (TermPF langs e) where
  lam = termPF . LamPF
  app f = termPF . AppPF f

instance EvalPF AbsPF where
  evalPF k (LamPF f) = lam $ k . f . termPF . VarPF
  evalPF _ (VarPF x) = x
  evalPF k (AppPF f x) = app (k f) (k x)
```

Applied Haskell

```haskell
pattern AsLam x <- TermPF SAbsPF (LamPF x)

instance PEval AbsPF where
  pevalPF k (AppPF f x) = case (k f, k x) of
                            (AsLam f', x'@(AsLit _)) -> f' x'
                            (f', x') -> app f' x'
  pevalPF k x = evalPF k x
```

Simplify, man

```
partialEval :: AllFinal MyLangPF e
            => (forall f.  AllFinal MyLangPF f
                => TermPF MyLangPF f a)
            -> TermPF MyLangPF e a
partialEval t = go t
  where go :: TermPF langs (TermPF langs e) a
           -> TermPF langs e a
        go (TermPF _ x) = pevalPF go x
```

Evaluating Partially Final

```
type MyLangPF = [ ArithPF, FancyPF, AbsPF ]

runEvalPF :: (forall e. AllFinal MyLangPF e
                => TermPF MyLangPF e a)
          -> a
runEvalPF t = runIdentity (go t)
  where go :: TermPF langs Identity b -> Identity b
        go (TermPF _ x) = evalPF go x
```

Remember testSum?

```
> runEvalPF testSum :: Int
5
```

Remove Tags

evaluate :: Initial -> Final

```
i2f :: TermPF langs e a -> e a
i2f (TermPF _ x) = evalPF i2f x
```

Let 'Er Rip!

```haskell
testProg :: (AbsF e, ArithF e) => e (Int -> Int)
testProg = lam $ \x -> add x (add (lit 2) (lit 3))
```

```
> putStrLn $ getCode testProg 1
(\x_1 -> (x_1 + (2 + 3)))
```

FINALLY!

```
peCode :: Code (Int -> Int)
peCode = i2f (partialEval testProg)
```

```
> putStrLn $ getCode peCode 1
(\x_1 -> (x_1 + 5))
```

Reduce 'Em If You Got 'Em

```haskell
testApp :: (AbsF e, ArithF e) => e Int
testApp = app (lam $ \x -> add x x) (lit 21)
```

```
> putStrLn $ getCode testApp 1
((\x_1 -> (x_1 + x_1)) 21)
```

Reduce 'Em If You Got 'Em

```
testApp :: (AbsF e, ArithF e) => e Int
testApp = app (lam $ \x -> add x x) (lit 21)
```

```
> putStrLn $ getCode testApp 1
((\x_1 -> (x_1 + x_1)) 21)
```

```
peApp :: Code Int
peApp = i2f (partialEval  testApp)
```

```
> putStrLn $ getCode peApp 1
42
```

An Emulator in an Optimizer

```
instance PEval FancyPF where
  pevalPF k (FancyOpPF x) = case k x of
                              AsLit x' -> litPF . runIdentity $
                                                fancyOp (lit x')
                              x' -> fancyOp x'
```

Calling the Hardware SDK

```haskell
testFancy :: (AbsF e, ArithF e, FancyF e)
          => e (Int -> Int)
testFancy = lam $ \x ->
            add x (fancyOp (add (lit 5) (lit 10)))
```

```
> putStrLn $ getCode testFancy 1
(\x_1 -> (x_1 + (hardwareOperation (5 + 10))))
```

Emulated Hardware

```
peFancy :: Code (Int -> Int)
peFancy = i2f (partialEval testFancy)
```

```
> putStrLn $ getCode peFancy 1
(\x_1 -> (x_1 + 57))
```