A Syntax for Composable Data Types in Haskell

A User-friendly Syntax for Solving the Expression Problem

Fredrik Albers
Anna Romeborn

Chalmers University of Technology

December 6, 2022

Introduction

```
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
```

Introduction

```
data Expr = Const Int | Add Expr Expr | Mul Expr Expr

eval :: Expr -> Int
eval (Const i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

Introduction

```haskell
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
          | Neg Expr

eval :: Expr -> Int
eval (Const i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

Introduction

```haskell
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
          | Neg Expr

eval :: Expr -> Int
eval (Const i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Neg e) = (-1) * eval e
```

Introduction

```haskell
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
          | Neg Expr

eval :: Expr -> Int
eval (Const i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Neg e) = (-1) * eval e
```

- Extend both data types and set of functions without recompiling old code

Introduction

```haskell
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
          | Neg Expr

eval :: Expr -> Int
eval (Const i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Neg e) = (-1) * eval e
```

- Extend both data types and set of functions without recompiling old code
- The expression problem

This Project

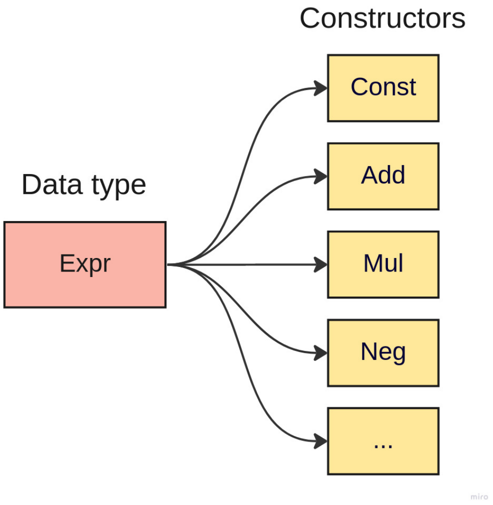
- Designed syntax components for a solution to the expression problem

This Project

- Designed syntax components for a solution to the expression problem
- A transformation into standard Haskell with `compdata`

Background

Automatic Extension of Data Types



Open Data Types

Automatic extension of data types

Open Data Types

Automatic extension of data types

```
open data Expr :: *

Const :: Int -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr
```

Open Data Types

Automatic extension of data types

```
open data Expr :: *

Const :: Int -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr

open eval :: Expr -> Int

eval (Const i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

Open Data Types

Automatic extension of data types

```
open data Expr :: *

Const :: Int -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr

open eval :: Expr -> Int

eval (Const i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2

Neg :: Expr -> Expr
eval (Neg e) = (-1) * eval e
```

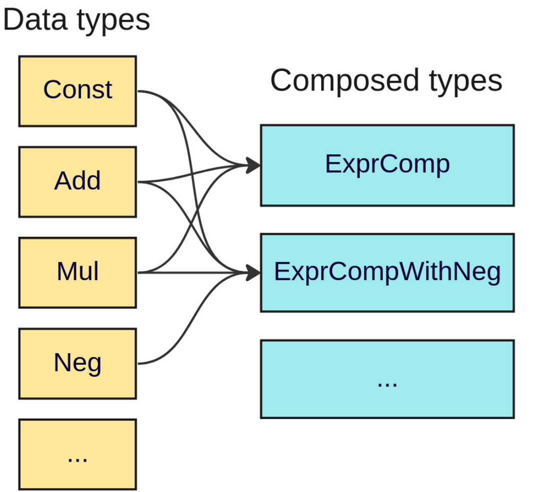
Open Data Types

Automatic extension of data types

```
open data Expr :: *

Const :: Int -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr

open eval :: Expr -> Int

eval (Const i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2

Neg :: Expr -> Expr
eval (Neg e) = (-1) * eval e
```

Simple syntax, but limited in expressive power

Composable Data Types



Data Types in compdata

Composable data types

Data Types in compdata

Composable data types

```
data Term f = In (f (Term f))

data (f :+: g) e = Inl (f e) | Inr (g e)
```

Data Types in compdata

Composable data types

```
data Term f = In (f (Term f))

data (f :+: g) e = Inl (f e) | Inr (g e)

data Const a = Const Int
data Op a = Add a a | Mul a a
```

Data Types in compdata

Composable data types

```haskell
data Term f = In (f (Term f))

data (f :+: g) e = Inl (f e) | Inr (g e)

data Const a = Const Int
data Op a = Add a a | Mul a a

type ExprComp = Term (Const :+: Op)
```

Data Types in compdata

Composable data types

```
data Term f = In (f (Term f))

data (f :+: g) e = Inl (f e) | Inr (g e)

data Const a = Const Int
data Op a = Add a a | Mul a a

type ExprComp = Term (Const :+: Op)

data Neg a = Neg a

type ExprWithNeg = Term (Const :+: Op :+: Neg)
```

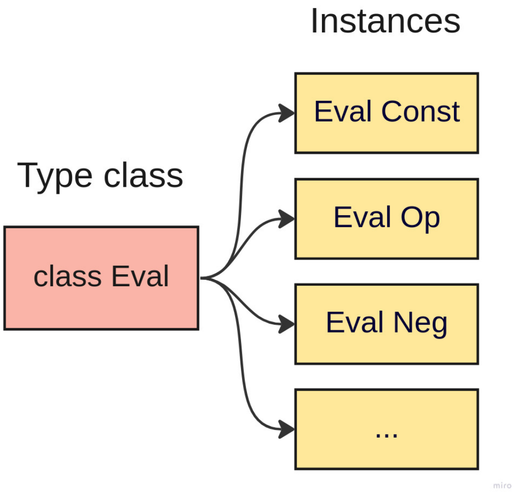
Data Types in compdata

Composable data types

```
data Term f = In (f (Term f))

data (f :+: g) e = Inl (f e) | Inr (g e)

data Const a = Const Int
data Op a = Add a a | Mul a a

type ExprComp = Term (Const :+: Op)

data Neg a = Neg a

type ExprWithNeg = Term (Const :+: Op :+: Neg)
```

Complex syntax, but higher expressive power

Automatic Extension of Functions



Extensible Functions in compdata

```
class Eval f where
    eval' :: Eval g => f (Term g) -> Int
```

Extensible Functions in compdata

```
class Eval f where
    eval' :: Eval g => f (Term g) -> Int

eval :: (Eval f) => Term f -> Int
eval = eval' . unTerm
```

Extensible Functions in compdata

```haskell
class Eval f where
    eval' :: Eval g => f (Term g) -> Int

eval :: (Eval f) => Term f -> Int
eval = eval' . unTerm

instance Eval Const where
    eval' (Const i) = i
instance Eval Op where
    eval' (Add e1 e2) = eval e1 + eval e2
    eval' (Mul e1 e2) = eval e1 * eval e2
```

Extensible Functions in compdata

```
class Eval f where
    eval' :: Eval g => f (Term g) -> Int

eval :: (Eval f) => Term f -> Int
eval = eval' . unTerm

instance Eval Const where
    eval' (Const i) = i
instance Eval Op where
    eval' (Add e1 e2) = eval e1 + eval e2
    eval' (Mul e1 e2) = eval e1 * eval e2

instance (Eval f, Eval g) => Eval (f :+: g) where
    eval' (Inl a) = eval' a
    eval' (Inr b) = eval' b
```

Extensible Functions in compdata

```
class Eval f where
    eval' :: Eval g => f (Term g) -> Int

eval :: (Eval f) => Term f -> Int
eval = eval' . unTerm

instance Eval Const where
    eval' (Const i) = i
instance Eval Op where
    eval' (Add e1 e2) = eval e1 + eval e2
    eval' (Mul e1 e2) = eval e1 * eval e2

$(derive [liftSum] [''Eval])
```

Extensible Functions in compdata

```
class Eval f where
    eval' :: Eval g => f (Term g) -> Int

eval :: (Eval f) => Term f -> Int
eval = eval' . unTerm

instance Eval Const where
    eval' (Const i) = i
instance Eval Op where
    eval' (Add e1 e2) = eval e1 + eval e2
    eval' (Mul e1 e2) = eval e1 * eval e2

$(derive [liftSum] [''Eval])


instance Eval Neg where
    eval' (Neg e) = (-1) * eval e
```

Constructors in compdata

```
threePlusFive :: ExprComp
threePlusFive = In (Inr (Add (In (Inl (Const 3)))
                             (In (Inl (Const 5))))))
```

Constructors in compdata

```
threePlusFive :: ExprComp
threePlusFive = In (Inr (Add (In (Inl (Const 3)))
                             (In (Inl (Const 5)))))
```

Subsumption

```
inject :: (g :<: f) => g (Term f) -> Term f
```

Constructors in compdata

```
threePlusFive :: ExprComp
threePlusFive = In (Inr (Add (In (Inl (Const 3)))
                             (In (Inl (Const 5))))))
```

Subsumption

```
inject :: (g :<: f) => g (Term f) -> Term f
```

Smart constructors

```
iAdd :: (Op :<: f) => Term f -> Term f -> Term f
iAdd x y = inject (Add x y)
```

Constructors in compdata

```
threePlusFive :: ExprComp
threePlusFive = In (Inr (Add (In (Inl (Const 3)))
                             (In (Inl (Const 5))))))
```

Subsumption

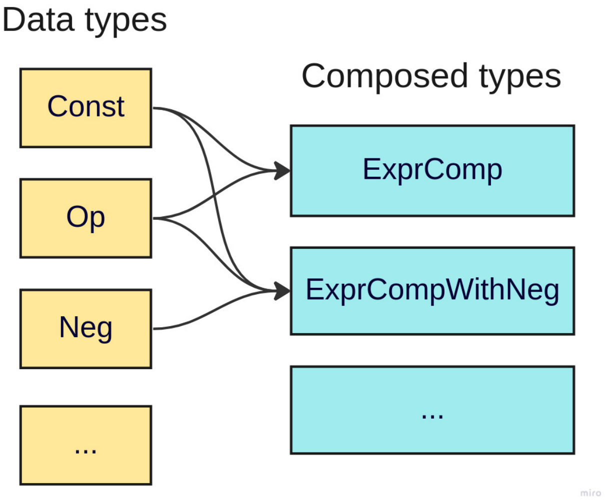
```
inject :: (g :<: f) => g (Term f) -> Term f
```

Smart constructors

```
iAdd :: (Op :<: f) => Term f -> Term f -> Term f
iAdd x y = inject (Add x y)

threePlusFive' :: ExprComp
threePlusFive' = iConst 3 `iAdd` iConst 5
```

Our syntax and transformation

Data Types


Data types
Composed types
Const
Op
Neg
...
ExprComp
ExprCompWithNeg
...
miro

Data Types

```
data piece Const = Const Int
data piece Op = Add Expr Expr
              | Mul Expr Expr
type ExprComp = (Const | Op)
```

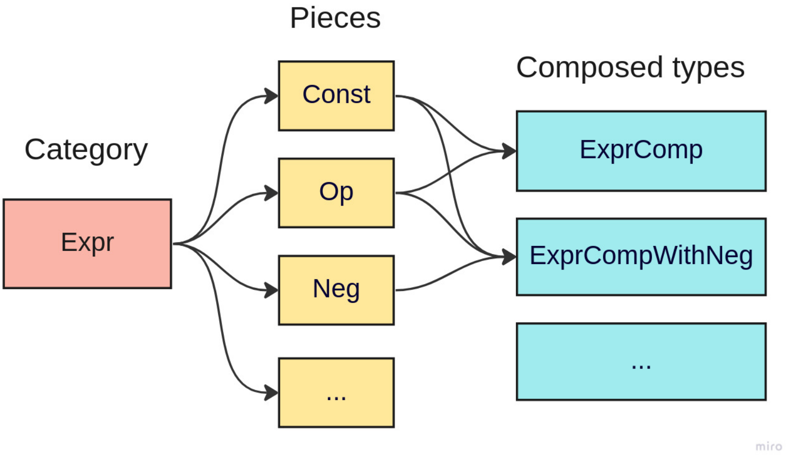
Data Types

```
piececategory Expr

data piece Expr ==> Const = Const Int
data piece Expr ==> Op = Add Expr Expr
                       | Mul Expr Expr

type ExprComp = Expr ==> (Const | Op)
```

Data Types

```
piececategory Expr

data piece Expr ==> Const = Const Int
data piece Expr ==> Op = Add Expr Expr
                       | Mul Expr Expr
type ExprComp = Expr ==> (Const | Op)

data piece Expr ==> Neg = Neg Expr

type ExprCompWithNeg = Expr ==> (Const | Op | Neg)
```

Data Types


Pieces
Composed types
Category
Expr
Const
Op
Neg
...
ExprComp
ExprCompWithNeg
...
miro

Transformation of Data Types

```
data piece Expr ==> Const = Const Int
data piece Expr ==> Op = Add Expr Expr
                       | Mul Expr Expr

type ExprComp = Expr ==> (Const | Op)
```

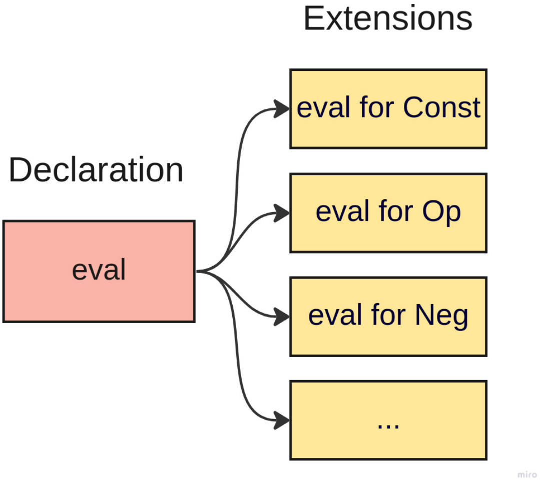
Transformation of Data Types

```
data piece Expr ==> Const = Const Int
data piece Expr ==> Op = Add Expr Expr
                       | Mul Expr Expr

type ExprComp = Expr ==> (Const | Op)
```

```
data Const a = Const Int
data Op a = Add a a | Mul a a
```

```
type ExprComp = Term (Const :+: Op)
```

Extensible Functions



Extensible Functions

```
eval -: Expr -> Int
```

Extensible Functions

```
eval -: Expr -> Int

ext eval for Const where
    eval (Const i) = i

ext eval for Op where
    eval (Add e1 e2) = eval e1 + eval e2
    eval (Mul e1 e2) = eval e1 * eval e2
```

Transformation of Function Declaration

```
eval -: Expr -> Int
```

Transformation of Function Declaration

```
eval -: Expr -> Int
```

```
class Eval f where
    eval' :: Eval g => f (Term g) -> Int
```

```
$(derive [liftSum] [''Eval])
```

```
class Eval_outer t where
    eval :: t -> Int
instance Eval g => Eval_outer (Term g) where
    eval = eval' . unTerm
```

Transformation of Function Extensions

```
ext eval for Const where
    eval (Const i) = i

ext eval for Op where
    eval (Add e1 e2) = eval e1 + eval e2
    eval (Mul e1 e2) = eval e1 * eval e2
```

Transformation of Function Extensions

```
ext eval for Const where
    eval (Const i) = i

ext eval for Op where
    eval (Add e1 e2) = eval e1 + eval e2
    eval (Mul e1 e2) = eval e1 * eval e2
```

```
instance Eval Const where
    eval' (Const i) = i
```

```
instance Eval Op where
    eval' (Add e1 e2) = eval e1 + eval e2
    eval' (Mul e1 e2) = eval e1 * eval e2
```

Constructors

```haskell
threePlusFive :: ExprComp
threePlusFive = Const 3 `Add` Const 5
```

Constructors

```haskell
threePlusFive :: ExprComp
threePlusFive = Const 3 `Add` Const 5
```

Smart constructors

Constructors

```haskell
threePlusFive :: ExprComp
threePlusFive = Const 3 `Add` Const 5
```

Smart constructors

```haskell
threePlusFive :: ExprComp
threePlusFive = iConst 3 `iAdd` iConst 5
```

Piece Constraints

```
constTwo :: Expr ==> (Const)
constTwo = Const 2

twoMulThreePlusFive ::
    Expr ==> (Const | Op)
twoMulThreePlusFive = constTwo `Mul`
    (Const 3 `Add` Const 5)
```

Piece Constraints

```
constTwo :: Const partof e => e
constTwo = Const 2

twoMulThreePlusFive ::
    (Const partof e, Op partof e) => e
twoMulThreePlusFive = constTwo `Mul`
    (Const 3 `Add` Const 5)
```

Piece Constraints

```
constTwo :: Const partof e => e
constTwo = Const 2

twoMulThreePlusFive ::
    (Const partof e, Op partof e) => e
twoMulThreePlusFive = constTwo `Mul`
    (Const 3 `Add` Const 5)
```

```
constTwo :: Const :<: f => Term f
```

```
twoMulThreePlusFive ::
    (Const :<: f, Op :<: f) => Term f
```

Transforming `e` to `Term f` is difficult!

Piece Constraints with PartOf

```haskell
class PartOf f e where
    inject' :: f e -> e
instance f :<: g => PartOf f (Term g) where
    inject' = inject
```

Piece Constraints with PartOf

```
class PartOf f e where
    inject' :: f e -> e
instance f :<: g => PartOf f (Term g) where
    inject' = inject
```

```
constTwo :: PartOf Const e => e
```

```
twoMulThreePlusFive ::
    (PartOf Const e, PartOf Op e) => e
```

Smart Constructors with PartOf

```haskell
iConst :: (PartOf Const e) => Int -> e
iConst i = inject' (Const i)
```

```haskell
iAdd :: (PartOf Op e) => e -> e -> e
iAdd e1 e2 = inject' (Add e1 e2)
```

```haskell
iMul :: (PartOf Op e) => e -> e -> e
iMul e1 e2 = inject' (Mul e1 e2)
```

Function Constraints

```
evalCond -: Expr -> Bool -> Int

evalFalse :: Expr ==> (Const | Op) -> Int
evalFalse a = evalCond a False
```

Function Constraints

```
evalCond -: Expr -> Bool -> Int

evalFalse :: e with evalCond => e -> Int
evalFalse a = evalCond a False
```

Function Constraints

```
evalCond -: Expr -> Bool -> Int

evalFalse :: e with evalCond => e -> Int
evalFalse a = evalCond a False
```

```
evalFalse :: EvalCond_outer e => e -> Int
evalFalse a = evalCond a False
```

Function Constraints

```
evalCond -: Expr -> Bool -> Int

evalFalse :: e with evalCond => e -> Int
evalFalse a = evalCond a False
```

```
evalFalse :: EvalCond_outer e => e -> Int
evalFalse a = evalCond a False
```

```
class EvalCond_outer t where
    evalCond :: t -> Bool -> Int
instance EvalCond g => EvalCond_outer (Term g) where
    evalCond = evalCond' . unTerm
```

Discussion and Conclusion

Multi-Variant Pieces

```
data piece Expr ==> Op = Add Expr Expr | Mul Expr Expr
```

Should a piece have several constructors?

Multi-Variant Pieces

```
data piece Expr ==> Op = Add Expr Expr | Mul Expr Expr
```

Should a piece have several constructors?

```
data piece Expr = Const Int
data piece Expr = Add Expr Expr
data piece Expr = Mul Expr Expr
type ExprComp = Expr ==> (Const | Add | Mul)
```

Multi-Variant Pieces

```
data piece Expr ==> Op = Add Expr Expr | Mul Expr Expr
```

Should a piece have several constructors?

```
data piece Expr = Const Int
data piece Expr = Add Expr Expr
data piece Expr = Mul Expr Expr
type ExprComp = Expr ==> (Const | Add | Mul)
```

```
ext eval (Const i) = i
ext eval (Add e1 e2) = eval e1 + eval e2
ext eval (Mul e1 e2) = eval e1 * eval e2
```

Multi-Variant Pieces

```
data piece Expr ==> Op = Add Expr Expr | Mul Expr Expr
```

Should a piece have several constructors?

```
data piece Expr = Const Int
data piece Expr = Add Expr Expr
data piece Expr = Mul Expr Expr
type ExprComp = Expr ==> (Const | Add | Mul)
```

```
ext eval (Const i) = i
ext eval (Add e1 e2) = eval e1 + eval e2
ext eval (Mul e1 e2) = eval e1 * eval e2
```

But more piece constraints, larger composed types,
no way to group constructors

Conclusion

Accomplishments:

- Characterizing some solutions to the expression problem

Conclusion

Accomplishments:

- Characterizing some solutions to the expression problem
- A syntax for composable data types
- A working transformation into standard Haskell

Conclusion

Accomplishments:

- Characterizing some solutions to the expression problem
- A syntax for composable data types
- A working transformation into standard Haskell
- Combined concepts of composable data types and automatically extended data types

The End

Thank you for your attention!