Pure Functional Encodings of the Expression Problem

Submitted in Partial Fullfillment of the Requirements for the Qualifying Exam

Alex Hubers
University of Iowa, Department of Computer Science

Abstract

Statically-typed programming languages face an expressiveness problem in extending data types. In functional languages, adding new cases to algebraic data types requires refactoring functions over that data type. For example, adding new constructs to an abstract syntax tree requires refactoring key functions (e.g, evaluation, parsing, and printing) over the tree. This leads to problems in reuse and modularity of code. This report focuses on approaches to address the expression problem in Haskell, a statically typed pure functional programming language. We principally consider encodings of extensible variants in Haskell used to define modular, extensible ASTs.

1 Introduction

In the expression problem [9], Wadler outlines a deficit in statically-typed programming language design: it is easy to add new functions over algebraic data types, but adding new cases to the data type requires refactoring all functions defined over it. Stated originally:

> The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety.

The expression problem manifests itself orthogonally in both functional and object-oriented data structures: In functional programming, it is easy to add functions over data types, but difficult to add cases. In object-oriented programming, it is easy to extend cases via subclasses, but difficult to add new methods to existing classes. This report will focus on the former, with an emphasis on demonstrating and addressing the expression problem by showing proposed methods to add new cases to abstract syntax trees (ASTs) in Haskell, a statically-typed pure functional language.

We will consider primarily the encodings of extensible variants in Haskell, and papers which trace their evolution. A variant is the dual of a record, meaning a type with labeled cases and just one case inhabited. Variants exist natively in Haskell, but they are closed to extension. We will show how the expression problem can be resolved then via the encoding of extensible variants, i.e, how to create extensible variants and thereafter extensible ASTs.

Roadmap We will first introduce the expression problem's benchmark problem in Section 2. We introduce solutions in chronological order, starting with Monad Transformers and Modular Interpreters [2] (MOD). Each solution extends previous work. MOD introduces a disjoint sum operator and a subtype inclusion class (Section 3), which is extended by Swierstra in Data Types à la Carte [8] to functors (Section 4). DTC is well regarded as a functional pearl that offers perhaps the most canonically recognized form of extensible variants in Haskell. In Section 5, we outline further improvements to this encoding via Variations on Variants [4] (VAR, Section 5). VAR uses instance chains to resolve problems in Swierstra's inclusion class. VAR additionally defines a branching operator for a less ad-hoc manner of defining case-wise functions over ASTs. We conclude with a discussion of these encodings feasibility and practicality, and where to go next.

2 The Expression Problem

To illustrate the expression problem, as well as familiarize the reader with Haskell, let us begin with a simple arithmetic language supporting addition and `Int` literals.

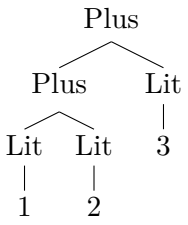
```
data Arith = Lit Int | Plus Arith Arith
```

In place of objects, Haskell has algebraic data types. Data type declarations begin with the `data` keyword, followed by the name of the type being declared. Here we are stating that the `Arith` data type has two constructors: `Lit`, for integer literals, and `Plus`, for expression addition. Constructors are separated by the `|` symbol. Arguments to constructors are listed positionally, e.g, the syntax `Plus Arith Arith` means that the constructor `Plus` expects two arguments of type `Arith`. The two constructors are thus term-level functions with the following type annotations:

```
Lit  :: Int -> Arith
Plus :: Arith -> Arith -> Arith
```

Type annotations can be given to definitions in Haskell via the `::` symbol. `Lit` and `Plus` are functions, and therefore have arrow-types. The arrows specify the type of expected arguments of each function. For example, `Lit :: Int -> Arith` says "give me an `Int` and I will return an `Arith`". This extends to multi-argument functions, like `Plus`. The `Arith` type is recursive, meaning the `Plus` constructor requires two terms already typed at `Arith` to build a `Plus` case. Thus we build the tree upwards, starting with `Lit`. Consider a small tree representing the expression `((1 + 2) + 3)`. Note that function application is positional on Haskell, e.g, we write `Plus (Lit 1) (Lit 2)` in place of `Plus(Lit(1), Lit(2))`.

```
smallExpr :: Arith
smallExpr = Plus (Plus (Lit 1) (Lit 2)) (Lit 3)
```



We can evaluate our small language by recursing over the AST. To do so, we pattern match on the constructors of `Arith`. The `eval` function below is typed at `Arith -> Int`, meaning it takes an `Arith` tree and produces an `Int` value. The first equation pattern matches on the leaf `Lit` nodes: when you reach a leaf, return `x :: Int`. The second equation says: when you are at a `Plus` branch, recurse and add the results.

```
eval :: Arith -> Int
eval (Lit x) = x
eval (Plus x y) = (eval x) + (eval y)

> eval smallExpr
  6
```

We can print our tree back to the expression syntax just as easily. To do so, we will use Haskell's `Show` typeclass. A typeclass in Haskell is a means of restricting a function's polymorphism to just those types that meet some interface. In this way they behave similarly to interfaces and generics in object-oriented languages like Java.

```
class Show a where
    show :: a -> String
```

The above reads that the type `a` satisfies the **`Show`** typeclass constraint if there exists function **`show`** typed at `a -> String`. The **`instance`** keyword expresses that `Arith` meets the interface of **`Show`** by providing a definition for **`show`** with `a = Arith`.

```
instance Show Arith where
  show (Lit x)    = show x
  show (Plus x y) = "(" ++ show x ++ " + " ++ show y ++ ")"

> show smallExpr
  "((1 + 2) + 3)"
```

The expression problem arises when adding new cases to our arithmetic AST. Suppose, as the simplest example, we wish to add support for multiplication. We can augment the `Arith` data type with a new constructor `Mult`,

```
data Arith = Lit Int | Plus Arith Arith | Mult Arith Arith
```

but we must now rewrite all functions over `Arith` (e.g, `eval` and **`show`**) to account for the new `Mult` case. This is to say, we may now express `(1 * 2)` via

```
three = Mult (Lit 1) (Lit 2)
```

but the terms **`show three`** and `eval three` are not defined. Accordingly, we may add the following lines back to `eval` and **`show`**, respectively. The aim in solving the expression problem is to not do so, however, and instead define these cases modularly.

```
eval (Mult x y) = (eval x) * (eval y)
show (Mult x y) = "(" ++ show x ++ " * " ++ show y ++ ")"
```

This example forms Wadler's benchmark [9] for addressing the expression problem, and will be our toy example through this report. The benchmark may seem artificial, but the lack of modularity is not: consider a number of abstract syntax trees used for interpreting a sufficiently expressive language (e.g, Haskell), which may have numerous intermediate forms. Not only do we face difficulty in extending such a language with new language features, but we also suffer a lack of component reuse: common constructs across multiple ASTs must be duplicated. As an extreme case, consider Leroy's three-person year verified compiler project [1] in Coq, which consists of eight intermediate languages in addition to the source and target languages, many of which are minor variations of each other. Thus, data type extensibility is desirable in compilation and interpretation of languages, as the intermediate data structures commonly have high overlap in patterns.

More broadly, The expression problem is also made salient when trying to extend data types from an external source. This is common practice in object-oriented software development: receive a superclass and extend it through object inheritance. The same cannot be done in Haskell without the extensible variant encodings we will now discuss.

3 First Steps: Monad Transformers and Modular Interpreters

In Monad Transformers and Modular Interpreters [2] (MOD), Liang et al. tackle this problem indirectly in the context of modular interpreters. Namely, they consider the modular combination of a language supporting numerous features, such as arithmetic operations, function bindings, variable assignment, tracing, and nondeterminism. To isolate the paper's contributions with respect to the expression problem, we will consider just their approach to the modular combination of an arithmetic component, as seen above.

The first step is to combine cases modularly, i.e, to define cases separately and glue them back together. To do so, the authors first define a disjoint sum operator, which they dub the *extensible union type*:

```
data OR a b = L a | R b
```

This type operator is more commonly known as the **Either** type today (MOD dates to 1995, actually preceding Wadler's posing of the expression problem in 1998). The `OR` data type is a type operator: it expects two type arguments, represented as type variables `a` and `b`, and returns a new type. For example, `OR Int String` would represent a type that can either be an **Int** or a **String**. If it is an **Int**, we inject into the left side via `L`; if it is a **String**, you inject into the right via `R`. We can use the `OR` type to modularly combine our arithmetic AST's constructors.

```
type Term = OR Val (OR Add Mult)
data Val  = Lit Int
data Add  = Plus Term Term
data Mult = Times Term Term
```

The **type** keyword in Haskell defines a type synonym; it does *not* define a new type in the same way that the **data** keyword does. Rather, `Term` is a convenient placeholder for the definition `OR Val (OR Add Mult)`. Note the mutual recursion in these definitions: we define `Term` as the right-nested sum of `Val`, `Add`, and `Mult`; we define the `Val` and `Add` summands with recursive subdata at type `Term`. Construction of terms at type `Term` thus becomes tricky – we will need to appropriately inject into the `OR` summands. We define a Subtype relation to help us inject more easily.

```
class Subtype sub sup where
  inj :: sub -> sup
instance Subtype a a where
  inj = id
instance Subtype a (OR a b) where
  inj = L
instance Subtype a b => Subtype a (OR c b) where
  inj a = R (inj a)
```

We define the `Subtype` relation through a multi-parameter typeclass. For one type to be a subtype of another, there must be an injection from `sub` to `sup`. We can give typeclass instantiations generically. The first instantiation states that every type injects into itself reflexively: `inj` is simply the identity function **id**. The next two instantiations express how to inject into an `OR` sum: first, we can inject `a` into `(OR a b)` by using the `L` constructor. Injection into deeper right-nested sums is given by the last instance, which says that if `a` is a subtype of `b` then we can inject `a` into `(OR c b)` by first injecting into `b` and then following the `R` route into `(OR c b)`.

With the `Subtype` relation declared, we can construct terms at type `Term` via "smart" constructors – meaning constructors which maintain some invariant. In our case, we want to build `Term`s while maintaining the invariant that they are wrapped by the appropriate `L` and `R` constructors. Smart constructors are given for **Int** literals, addition, and multiplication below.

```
lit :: Int -> Term
lit x = inj (Lit x)
plus :: Term -> Term -> Term
plus x y = inj (Plus x y)
times :: Term -> Term -> Term
times x y = inj (Times x y)
```

Because `inj` is defined generically for injection into `OR` types (which `Term` is), the overloaded `inj` operator appropriately injects into the correct summands. For example, the `inj` in `inj (Lit x)` matches the `Subtype a (OR a b)` typeclass instance, and therefore `inj` is simply `L`.

The next goal is to define evaluation for each case and combine the evaluators modularly. This is to say, we should be able to define evaluation piece-wise for the `Val`, `Add`, and `Mult` cases, and then use one generic evaluator to evaluate the `Term` type. The authors create a typeclass `InterpC` which specifies a case's evaluation.

```
class InterpC t where
```

```
  interp :: t -> Int
instance (InterpC t1, InterpC t2) => InterpC (OR t1 t2) where
  interp (L t) = interp t
  interp (R t) = interp t
```

The `InterpC` class specifies how a component is evaluated, with type variable `t` as placeholder for the component. The `InterpC` instantiation for `(OR t1 t2)` has a typeclass restriction, written as `(InterpC t1, InterpC t2) =>`, that `t1` and `t2` also satisfy the `InterpC` constraint. Thus the instantiation says that if we can interpret both `t1` and `t2`, then we can interpret `(OR t1 t2)`. For example, if we can interpret `Val` and `Add` components, then we can interpret `OR Val Add` by delegating appropriately to these components' `InterpC` instances. What is left is to define such instances for each case.

```
instance InterpC Val where
  interp (Lit x) = x
instance InterpC Add where
  interp (Plus x y) = interp x + interp y
instance InterpC Mult where
  interp (Times x y) = interp x * interp y
```

Integer literals evaluate trivially to the `Int` they contain. The `Add` and `Mult` cases hold subdata at type `Term`, i.e, `x` and `y` in each case have type `Term`. This means the calls `interp x` and `interp y` are not recursive calls, but rather to `interp` as defined for the `Term` type.

With these definitions in order, we can now evaluate an expression containing all three components, For example, `(1 + 2) * 3`:

```
nine :: Term
nine = (times (plus (lit 1) (lit 2)) (lit 3))
> interp nine
  9
```

This is a good first step, and forms the general strategy for the encodings which follow in this report. We first define our AST components individually. Then, we combine them modularly via a type operator. Finally, we define a typeclass for evaluation. Because the `InterpC` typeclass is well defined for each component, and `OR` sums also define interpretation via delegation to cases, the final `Term` type is evaluatable. This approach is certainly modular, but not extensible. In particular, we rely on the `Term` type synonym to define the type of recursive subdata. Recall that we supplied in advance the type of subdata to be `Term = OR Val (OR Add Mult)`. Suppose we wish to add a `Div` operator to our arithmetic language: we can provide the evaluator for `Div` modularly, but must add `OR Div` into the `Term` definition.

4 Data Types à la Carte

Our next goal is to permit extensibility in our final `Term` definition. In Data Types à la Carte [8], Swiestra extends the work of Liang et al. with a better component sum operator (e.g, the `OR` operator) and inclusion class (e.g, the `Subtype` class). The issue in extending the `Term` type is that the final AST's subdata is fixed at point of definition. For example, the arithmetic AST had subdata at type `OR Val (OR Add Mult)`. It follows that we wish to un-fix this subdata, which we can do by representing our components functorially.

4.1 Functorial Representations of Data Types

The key to modularity is to abstract away the recursive occurrences of the type in our AST declarations with type variables. By example:

```
data Val  x = Lit Int
data Add  x = Plus x x
```

```
data Mult x = Times x x
```

Each of these data types form functors. Within the context of typed functional programming, a functor is two components: a type-level operator `f`, which takes type `a` and builds `f a`; and a term-level operator (called `fmap`), which takes a function typed at `(a -> b)` and returns a function typed at `(f a -> f b)`. In Haskell, the **`Functor`** typeclass provides an interface for specifying the `fmap` function.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The above states that the type variable `f` satifies the **`Functor`** typeclass constraint if one can populate the term `fmap` at type `f` for any types `a` and `b`. For example, we can show the `Add` operator to be a functor via the following instantiation:

```
instance Functor Add where
  fmap f (Plus x y) = Plus (f x) (f y)
```

Intuitively, `fmap` means to map the function `f` over the functor's structure. Lists are perhaps the most known example of functors. For illustration, to map a function over a list functor is to apply `f` to all of its elements, e.g, below we map the function `(+1)` over a list to add one to each of the list's elements.

```
> fmap (+1) [1, 2, 3]
  [2, 3, 4]
```

The process of separating a recursive data type into its functorial structure is referred to by Sheard and Pasalic [7] as Two-Level Types: we have the structure operator (also called a signature functor), e.g `Add`, which gives us the shape of the type in the form of a functor. If desired, we can recover the recursive type by wrapping it back into itself, giving it the aforementioned two levels.

```
data Add' = Wrap (Add Add')
unwrap :: Add' -> Add Add'
unwrap (Wrap x) = x
```

The data type `Add'` "wraps" the functor `Add` back into itself by giving the recursive type occurence `Add'` as argument to the `Add` functor. This can be generalized not just to `Add` but to all of our functors via the least fixed-point operator `Expr`.

```
data Expr f = In (f (Expr f))
out :: Expr f -> f (Expr f)
out (In x) = x
```

The `Expr` operator is called a least fixed-point operator because it generates a type `T` such that `F T` and `T` are isomorphic. For our purposes, it is sufficient to see that the `Expr` operator generalizes `Add'`; it takes a functor and replaces all occurences of the type variable with the type itself. For example, `Expr Add` has subdata at type `Expr Add`.

4.2 Combining Signature Functors with Coproducts

The `OR` type operator forms a disjoint sum between two concrete types. However, use of the `OR` operator in combining cases required fixing the `Term` AST in advance. For example, to represent the expression `((1 + 2) * 3)`, we had to specify that the subdata of our AST had type `OR Val (OR Add Mult)`. With functorial representations of the `Add`, `Val`, and `Mult` types, the subdata is un-fixed. We lift the `OR` type operator to functors via the functor coproduct operator `:+:`.

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

The coproduct operator takes two functors `f` and `g` and produces a new functor `(f :+: g)` with constructor `Inl` when you wish inject into the left side via `f`, and `Inr` for the right, using `g`. The coproduct permits us to fuse signature functors back into abstract syntax trees without regard for the subdata type. For example, the type `Expr (Val :+: Add)` is isomorphic to the `Arith` tree:

```
data Arith = Lit Int | Add Arith Arith
```

What separates it from `Arith` is that it is rather miserable to write. Swierstra gives the following term representing the simple expression `(118 + 1219)` as example.

```
addExample :: Expr (Val :+: Add)
addExample = In (Inr (Add (In (Inl (Lit 118))) (In (Inl (Lit 1219)))))
```

4.3 Tidying Term Construction With Functor Subtyping

Like with the `OR` operator, we will also want smart constructors to tidy term construction. We want to build terms at type `Expr f` while maintaining the invariant that they are wrapped by the appropriate constructors `In`, `Inl`, and `Inr`. We lift MOD's `Subtype` relation to functors as follows. Note that the type operator `:<:` is used in infix position.

```
class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a
```

The intuition is that the functor `sub` is a functor subtype of `sup` if there exists an injection for all terms at type `sub a` into type `sup a`. For example, consider `(Val :<: (Val :+: Add))`. Given a term of type `Val a`, you can inject into `(Val :+: Add) a` via the `Inl` constructor.

As with MOD, Swiestra gives instantiations of the functor subtype operator in Figure 1, left. The first case states that every functor injects into itself trivially; the second case follows the left route, and the third case covers injection of `f` into the right functor `g` in both the case where `f = g` and the case that `g` is some larger coproduct. Large coproducts must be explicitly right-nested to ensure the third typeclass instance is matched. For example, we cannot infer that `Val` injects into the left nested instance `((Val :+: Add) :+: Mult)`. We will address this later in Section 5.

```
-- reflexivity
instance Functor f => f :<: f
  where
    inj = id
-- Left route
instance (Functor f, Functor g) =>
  f :<: (f :+: g)
  where
    inj = Inl
-- Right route
instance (Functor f,
          Functor g,
          Functor h,
          f :<: g) => f :<: (h :+: g)
  where
    inj a = Inr (inj a)
```

```
inject :: (g :<: f) =>
          g (Expr f) -> Expr f
inject d = In (inj d)
lit  :: (Val :<: f) =>
        Int -> Expr f
lit x = inject (Lit x)
plus :: (Add :<: f) =>
        Expr f -> Expr f -> Expr f
plus x y = inject (Plus x y)
times :: (Mult :<: f) =>
         Expr f -> Expr f -> Expr f
times x y = inject (Times x y)
```

Figure 1: Functor subtype instantiations (left) and smart constructors (right).

The `:<:` type operator also permits us to write our smart constructors as we initially desired (Figure 1, right). In the type signatures for the smart constructors, you can imagine `f`

as being some larger coproduct, e.g, in `addExample` below, the constructor `lit` has `f` instantiated with `(Val :+: Add)`. It follows that `plus` has the same instantiation. This permits both `lit` and `add` to construct the type `Expr (Val :+: Add)`. Uniformity becomes key in overcoming the subdata problem of expressing `((1 + 2) * 3)`. In this case, subtrees are at type `Expr (Val :+: (Add :+: Mult))`, and can therefore be combined as desired.

```
addExample :: Expr (Val :+: Add)
addExample = plus (lit 118) (lit 1219)

simpleExpr :: Expr (Val :+: (Add :+: Mult))
simpleExpr = times (plus (lit 1) (lit 2)) (lit 3)
```

4.4 Modular Evaluation With Folds

We can modularly define and combine syntactic constructs via signature functors, functor coproducts, and functor subtyping. The expression problem requires we need not rewrite the functions over these data types. Ideally, we modularly define functions such as `eval` for each signature functor. The key to doing so is to first observe that evaluation is a fold. A fold takes a recursive data type and reduces it to a value of another type. Consider the type of `fold` for lists:

```
fold :: (a -> b -> b) -> b -> [a] -> b
fold op b []       = b
fold op b (x : xs) = op x (fold op b xs)
```

The function `op :: (a -> b -> b)` combines a value of the list with the result of previous `op` applications until a value of type `b` is what remains. For example, consider the summation of a list using `(+) :: Int -> Int -> Int`.

```
> fold (+) 0 [1, 2, 3]
  6
```

Folding generalizes to any functorial data type. We've shown this in the case of trees; recall our evaluation of the `Arith` AST:

```
> eval (Plus (Plus (Lit 1) (Lit 2)) (Lit 3))
  6
```

When defining `fold` or `eval` over known functors, we utilize pattern matching to distinguish behavior based on cases. Generalizing the `fold` definition to arbitrary functor `f` requires further abstraction. Swiersta defines folds over the `Expr f` type as `foldExpr`. This formulation is known generically as a catamorphism [3].

```
foldExpr :: Functor f => (f a -> a) -> Expr f -> a
foldExpr f (In t) = f (fmap (foldExpr f) t)
```

The term typed at `(f a -> a)` is called an F-Algebra and can be thought of as giving the recursive step of evaluation for a given signature functor. The type variable `a` denotes the target of evaluation. For example, evaluation of the `Add` component with target `Int` is typed as the F-Algebra `Add Int -> Int`:

```
addAlg :: Add Int -> Int
addAlg (Plus x y) = x + y
```

4.4.1 Combining the Evaluators

Swiersta encapsulates evaluation of signature functor with the `Eval` typeclass. The strategy is to modularly define evaluation for each functor, then define one typeclass instantiation for `Eval` at type `f :+: g`. Our target language is arithmetic, so we fix `Int` as the target of evaluation.

```
class Functor f => Eval f where
  evalAlgebra :: f Int -> Int
```

Instantiation of **Eval** for functor coproducts can be inferred by delegating to the appropriate instantiation of `evalAlgebra` for `f` and `g`. Note that the magic of Haskell typeclasses is at work here: the `evalAlgebra` equations are *not* recursive. Rather, the `evalAlgebra` function occurring on the right-hand side of each equation is an instantiation with respect to that functor's typeclass satisfication. For example, `evalAlgebra x` in the `(Inl x)` case below uses the `evalAlgebra` defined for `f`, whereas `evalAlgebra y` uses the `evalAlgebra` defined for `g`.

```
instance (Eval f, Eval g) => Eval (f :+: g) where
  evalAlgebra (Inl x) = evalAlgebra x
  evalAlgebra (Inr y) = evalAlgebra y
```

What remains is to show that each syntactic component satisfies the **Functor** and **Eval** typeclass constraints. Example instantiations are given for the `Add` component below. The other cases follow an expected pattern.

```
instance Functor Add where
  fmap f (Plus e1 e2) =
    Plus (f e1) (f e2)

instance Eval Add where
  evalAlgebra (Plus x y) = x + y
```

The final `eval` term combines our efforts to provide one evaluation function for extensibly combined signature functors. Observe:

```
eval :: Eval f => Expr f -> Int
eval expr = foldExpr evalAlgebra expr

simpleExpr :: Expr (Mult :+: (Val :+: Add))
simpleExpr = times (plus (lit 1) (val 2)) (val 3)
> eval simpleExpr
  9
```

As desired, evaluation is defined for the fixed point of any functor `f` such that `f` is evaluatable. This means we do not need a `Term` type floating around. To demonstrate this flexibility, consider the types for the following terms:

```
three :: (Add :<: f, Val :<: f) => Expr f
three = plus (val 1) (val 2)
nine :: (Times :<: f, Val :<: f) => Expr f
nine = times (val 3) (val 3)

twelve :: Expr (Val :+: (Add :+: Mult))
twelve = eval (plus three nine)
```

The types of `three` and `nine` are suitably generic – they can fit into `Expr f` for any `f` into which their components inject. We can defer a concretization until evaluation, at which point we specify that `twelve` is the least fixed-point of the `(Val :+: (Add :+: Mult))` functor. So this addresses our problem with the MOD encoding: we can modularly add further cases, so long as the final evaluation type has the correct components.

4.5 The Problems With Inclusion

In practice, the generality of DTC can also lead to problems. Consider our representation of the simple expression `(1 + 2)` as `three` above. What type should this term have at point of evaluation? We can not actually evaluate this term at its most general type.

```
three :: (Add :<: f, Val :<: f) => Expr f
three = plus (val 1) (val 2)

> eval three
  error:
    Ambiguous type variable ‘f’ arising from a use of ‘eval’
    prevents the constraint ‘(Eval f)’ from being solved.
```

GHC cannot infer which concrete `f` to choose from. All of the types below are perfectly valid candidates for `Expr f`:

```
type E1  = Expr (Val :+: Add)
type E1' = Expr (Add :+: Val)
type E2r = Expr (Mult :+: (Val :+: Add))
type E2l = Expr ((Mult :+: Val) :+: Add)
type E3  = Expr (Val :+: (Val :+: Add))
```

Types `E1` and `E1'` are simply a commutation of the cases; `E2r` and `E2l` are separate associations of cases, and `E3` shows that redundant cases are permitted. It follows from `E3` that there are an infinite amount of types we may assign to even the constant `val 1`, which is problematic. Ideally, redundant cases are excluded. Additionally, DTC views `E2r` and `E2l` as distinct, and will not permit injection of `(Mult :+: Val)` into the left hand side of `E2l` (recall Figure 1; we only define injection into coproducts on the right hand side).

We isolate from types `E2l` and `E3` two deficiencies in DTC's inclusion class. First, terms at a type with the `Mult`, `Val`, and `Add` cases should be able to inject into left-hand summands. Second, types like `E3` with redundant cases should be rejected. More broadly, the goal is to define an `eval` function by cases, where expressions in any valid expression language above can be evaluated. That is to say, we should be able to type `nine = times three three` at its most general and evaluate it when coerced to both types `E2r` and `E2l`, and we should reject `E3`'s validity.

4.6 The Problems With Evaluation

Evaluation is defined as a fold for any functor `f` that satisfies the **`Eval`** typeclass. This yields two problems. First, it follows that additional functions over your data type (e.g, printing, parsing) will require additional typeclasses. This approach is taken in Two-Level Types and Generic Unification [7], but it is a lot of scaffolding to write what is otherwise a simple collection of functions. Take as example a **`print`** function to render your AST back to a string:

```
class Print f where
  print :: f Int -> String
```

Any additional functions will require such a class, and each function must be instantiated for each syntactic component. This is inconvenient, as we tend to write many functions over AST's.

The other problem with Swierstra's **`Eval`** typeclass is that it fixes **`Int`** as the carrier of the `evalAlgebra :: f Int -> Int` function. This means the evaluation target of each syntactic component must be **`Int`**. It may be the case, however, that syntactic components have distinct evaluation targets. These cannot intermingle. So a new **`Eval`** typeclass must be created for each evaluation target, or the **`Eval`** typeclass must be rewritten to be indexed by some type variable `v`.

Both of these restrictions elucidate a certain inflexibility of the typeclass approach to destructing your ASTs. It would be desirable for the programmer to be able to state their functions simply by cases. For example, we may want to state that the evaluation of our arithmetic language is the combination of the evaluators for addition, constants, and literals.

5 Variations on Variants

Variations on Variants [4] (VAR) addresses the problems above by extending DTC with a more expressive inclusion class that permits both left-nesting and right-nesting coproduct construction, and excludes coproducts with repeated types. Additionally, it defines a case subtraction typeclass and a generic expression-level branching combinator (denoted as the operator `(?)`) in place of an `Eval` typeclass. The former addresses our inclusion problem; the latter permits evaluation handlers to be defined as cases and combined modularly. For example, with the `?` operator we may write

```
evalArith = cases (evalMult ? (evalAdd ? evalVal))
```

to specify each branch of evaluation. This is a more usable interface for the programmer than the typeclass approach of DTC. We will discuss both of these improvements now.

5.1 A New Inclusion Class

VAR uses a type-directed version of injection, allowing uniform expression of terms of the various expression languages. The goal is to define a new inclusion operator, $\hookleftarrow$, which can type `three` at any of the types above and inject into any of the valid types above. To do so, we will use instance chains [5], which are a proposed extension to the Haskell class system. Instance chains permit more expressive typeclass resolution by letting cases be written in an if/then/else paradigm.

By way of example, we first define an `In` class to specify when a functor is a included in a coproduct (Figure 2). We write `In` as an infix operator using backticks, and will redefine our fixed-point and functor coproduct operators as `Fix` and $\oplus$ to separate them from the operators of DTC. Note that the `In` keyword is used twice: at the term-level, it is the constructor of the `Fix` type; at the type-level, it is a membership test for inclusion. Haskell permits type and term level operators to share names, as which one is in use can be inferred by context.

```
data Fix f = In (f (Fix f))
data (f ⊕ g) e = Inl (f e) | Inr (g e)

class In f g

instance f `In` f
else f `In` (g ⊕ h) if f `In` g
else f `In` (g ⊕ h) if f `In` h
else f `In` g fails
```

Figure 2: Membership test for coproducts.

The typeclass `In` has no functions and is simply a relation between functors `f` and `g`. The instance chain permits this idiom of defining typeclass instances by cases. The semantics are as expected: check the first case. If it passes, resolve; otherwise, check the next case. The base case states that every `f` may inject reflexively. If this is not matched, we check the left and right hand sides (`g` and `h`, resp.) recursively. If `f` cannot be found in either side, we fail; failure of typeclass resolution can be matched as a clause in other instance chains. We now use the `In` typeclass to direct the inclusion operator $\hookleftarrow$. Unlike DTC, this inclusion operator guides when cases should be met and resolved.

```
class f ⋐ g where
  inj :: f e -> g e

instance f ⋐ f where
  inj = id
else f ⋐ (g ⊕ h) if f ⋐ g, f In h fails where
  inj e = Inl (inj e)
else f ⋐ (g ⊕ h) if f ⋐ h, f In g fails where
  inj e = Inr (inj e)
else f ⋐ g fails
```

Figure 3: Overloaded injection function.

The ⋐ class is defined similarly to the inclusion operator `:<:` of DTC, but with the conditions necessary to permit left coproduct injection and reject redundant cases. The instance chain first specifies that `f` injects into itself directly. If this case is not matched, it checks if `f` injects into `g`; if so, inject into the left hand side. If not, check the right hand side, `h`. If `f` is in neither the left or right, we fail.

To address redundant cases, we insist that `f` not occur in the other summand in both the left and right hand injections. For example, for `f` to inject into `g` in `g ⊕ h`, we insist `f` is not in `h`. This excludes coproducts with redundant cases, e.g, `Val ⊕ Val`. Injection into left hand summands follows trivially from the second case: if `f` injects into `g`, then inject into the left hand side of `g ⊕ h`. Note that instance chains permit both of these features: redundant cases are excluded via the failure of `In` clauses, and instance chains permit the pattern `f ⋐ (g ⊕ h)` to have two cases (GHC rejects instance patterns occurring more than once).

The most general type of `three` is now

```
(Add ⋐ f, Val ⋐ f) => Expr f
```

Unlike in DTC, this term can be typed at both `E2l` and `E2r`, as directed typeclass resolution of ⋐ permits `Add` and `Val` to be found in `E2l`. Further, we cannot coerce `three` to the problematic type `E3`, as desired.

5.2 Branching

Now, our goal is to define a branching operator `(?)` that permits `eval` to be specified by cases. We first introduce a primitive branching operator (▽) which specifies the branching behavior of coproducts.

```
(▽) :: (f e -> a) -> (g e -> a) -> (f ⊕ g) e -> a
(f ▽ g) (Inl x) = f x
(f ▽ g) (Inr x) = g x
```

The ▽ operator may look familiar; it is defined similarly to how Swierstra branches evaluation of coproducts into left and right cases for the coproduct instantiation of the **`Eval`** typeclass (see Section 4.4.1). You can parse the ▽ type signature as thinking of `e` as your expression type and `a` as the target of evaluation. Then ▽ simply states: If you tell me how to consume an `f e` and a `g e` into target `a`, then I can consume `(f ⊕ g) e` into an `a` term. For example, consider evaluation again for our arithmetic language:

```
evalVal :: Val Int -> Int
evalVal (Lit x) = x

evalAdd :: Add Int -> Int
evalAdd (Plus x y) = x + y
```

These are precisely the F-Algebras we give in Swierstra's `Eval` typeclass instantiations for `Val` and `Add`, respectively. The difference is now we many define an eval just for these cases:

```
evalArith :: Fix (Val ⊕ Add) -> Int
evalArith = foldExpr (evalVal ▽ evalAdd)

> evalArith (plus (lit 1) (lit 2))
  3
```

The ▽ operator is primitive by design; we still must rely on Swierstra's `foldExpr` catamorphism to turn `(evalVal ▽ evalAdd)` into a proper fold. We next define an overloaded branching operator (?). The expression `m ? n` defines a function on coproducts types where `m` describes the behavior on the subtracted summand, and `n` describes the behavior on the remainder of the coproduct. We will introduce now a type level operator ⊖ for case subtraction as well as the definition of `?`.

```
class f ⊖ g = h where
  (?) :: (g e -> a) -> (h e -> a) -> f e -> a
```

The subtraction class ⊖ will hold if subtracting the `g` case from `f` results in `h`. As illustration, we would expect the following to hold:

```
(Val ⊕ Add) ⊖ Val = Add
((Val ⊕ Add) ⊕ Mult) ⊖ Add = Val ⊕ Mult
```

Typeclass instantiations for the case subtraction class are given in Figure 4.

```
instance (f ⊕ g) ⊖ f = g where
  m ? n = m ▽ n
else (f ⊕ g) ⊖ g = f where
  m ? n = n ▽ m
else (f ⊕ g) ⊖ h = (f ⊖ h) ⊕ g if h `In` g fails where
  m ? n = (m ? (n . Inl)) ▽ (n . Inr)
else (f ⊕ g) ⊖ h = f ⊕ (g ⊖ h) if h `In` f fails where
  m ? n = (n . Inl) ▽ (m ? (n . Inr))
```

Figure 4: Overloaded branching combinator.

The first case simply uses the ▽ operator to define ?. This is best understood by revisiting our simple case: `(Val ⊕ Add) ⊖ Val = Add`. We would expect that the `m :: Val Int -> Int` and `n :: Add Int -> Int` in (?) to specify the behavior of the `Add` and `Val` cases, respectively. This is precisely what ▽ does. Evaluators `eval1` and `eval2` are thus equivalent.

```
eval1, eval2 :: (Val :+: Add) Int -> Int
eval1 = (evalVal ▽ evalAdd)
eval2 = (evalVal ? evalAdd)
```

The second case subtracts `g` from `f ⊕ g` to get `h`, and `(?)` is simply the flip of the primitive branching operator. The last two cases are more intricate. The left-recursive case describes the case when `h` is a component of `f`, e.g,

```
((Val ⊕ Add) ⊕ Mult) ⊖ Val = ((Val ⊕ Add) ⊖ Val) ⊕ Mult
```

Note that we insist in this case that subtracted component (e.g, `Val`) does not occur in the right hand summand (e.g, `Mult`); that would be a right-recursive case, which is handled in the last **`else`** statement in the instance chain. Let use take the third case step by step.

Recall that we are given `m :: h e -> a` (the branching behavior of the subtracted component), and `n :: (f ⊖ h) ⊕ g -> a` (the branching behavior of the remainder), and wish to return a term typed at `(f ⊕ g) e -> a`. If the input value is of type `f`, then it is either of

type `h` or of type `f ∈ h`. This follows from `h` occurring in `f`. If it is of type `h`, it is handled by `m`. If it is of type `f ∈ h`, then it is handled by the left summand of `n`. Thus, the `f` case is handled by `m ? (n . Inl)` (where `.` denotes function composition). Alternatively, the input could be of type `g`. Then we know that the right summand of `n` handles the case, i.e, `n . Inr`. The right-recursive case is parallel to the left, so elaboration is omitted.

We now demonstrate how the the `(?)` operator achieves our goal of providing a modular evaluator by cases sans typeclasses like **`Eval`**. We give evaluation functions for each case, but bind the recursive step to variable `r`. Each evaluator must produce the same type, which is **`Int`** here, but need not be in general.

```
evalVal (Lit x) r      = x
evalAdd (Plus x y) r   = r x + r y
evalMult (Times x y) r = r x * r y
```

We will populate the recursive `r` placeholder in the `cases` helper:

```
cases cs = f where f (In c) = cs c f

eval1 :: (f ∈ Val = Add) => Fix f -> Int
eval1 = cases (evalVal ? evalAdd)
```

The `cases` helper unrolls the `Fix` data type to `c` and passes this to `cs`. Note that `f` is given to `cs` as the recursive `r`, i.e, the recursive step is truly passed along to our evaluators. The `eval1` evaluator can evaluate languages at type `E1` or `E1'` (i.e, languages which support the `Val` and `Add` cases). We extend our evaluator to support multiplication easily.

```
eval2 = cases (evalMult ? (evalAdd ? evalVal))
```

5.3 Example: Desugaring

The cases approach we have taken overcomes two problems we identified with the **`Eval`** typeclass. We are able to define multiple functions (not just evaluation) easily, and we have un-fixed the evaluation target of our algebras. This is to say, we might write a printer as

```
print = cases (printMult ? (printAdd ? printVal))
```

Another benefit to this approach is that we may desugar syntactic constructs into more primitive ones. Suppose, for example, you had a `Square` signature functor to denote integer squaring:

```
data Square e = Sqr e
```

We can desugar a term at type `Square e` to `Mult e e` as folows:

```
inj' x = In (inj x)

desugarSqr :: (f ∈ Square = g, Mult ∈ g, Functor g) =>
  Fix f -> Fix g
desugarSqr = cases (sqr ? def) where
  sqr (Square e) r = inj' (Times (r e) (r e))
  def e r = In (fmap desugarSqr e)
```

The locally defined `sqr` function specifies how a `Square e` term can be transformed and properly rewrapped into a `Mult` term. The `def` function specifies default behavior.

5.4 DTC & VAR: Discussion and Critique

5.4.1 Concretization

Consider again the expression `(1 + 2)`. To evaluate this term, we require a concrete type, but to give such an annotation leads to problems in modularity. Suppose we choose the most sensible concrete option (Figure 5, right). Most general types are given on the left.

```
three :: (Val ⊲ f, Add ⊲ f) => Expr f
three = plus (val 1) (val 2)

nine :: (Val ⊲ f, Add ⊲ f, Mult ⊲ f)
  => Expr f
nine = times three three
```

```
three :: Fix (Val ⊕ Add)
three = plus (val 1) (val 2)

nine :: Fix (Mult ⊕ (Val ⊕ Add))
nine = times three three
```

Figure 5: Assigning types to simple terms; most general on left, concrete on right.

The type given to `nine`, while looking sensible enough, does not typecheck; the `times` smart constructor requires that `Mult` injects into `(Val :+: Add)`, meaning `times three three` is ill-typed. So we must either: A) reassign to either `three` or `nine` their most general type; B) assign both `three` and `nine` their most general types and defer concretization; or C) assign to both `three` and `nine` the concretization `Fix (Val ⊕ (Add ⊕ Mult))`. Approach B seems the most reasonable, but imposes a burden upon the programmer of providing explicit most-general type annotations to each term up until point of evaluation. This is the de facto norm of both DTC and VAR. The problem is compounded in that GHC is unable to infer these most general types in DTC.

A fourth approach might be to provide a smarter-er injection function which can inject `three` at type `(Val ⊕ Add)` into `(Mult ⊕ (Val ⊕ Add))`. Then we can define `nine` as follows.

```
inject' :: g ⊲ f => Expr g -> Expr f
inject' (In e) = In (inj (fmap inject' e))

nine :: Fix (Val ⊕ (Add ⊕ Mult))
nine = In (inj ((inject' three) `Times` (inject' three)))
```

However, this is silly for a few reasons. First, native GHC (via the DTC approach) will not resolve that `(Val :+: Add) :<: (Val :+: Add)`, as it sees this as both the reflexive instantiation of `:<:` as well as the left route. That is to say, it sees this as both `f :<: f` and `f :<: (g :+: h)`. GHC will simply see these cases as overlapping and fail to pick either. With instance chains, the instances are ordered conditionally, meaning VAR would resolve the reflexive case first.

Second, the `inject'` function is operationally bizarre. Recall that a term at type `Expr g` may be some recursive type. Suppose it is a tree, e.g, the `Add` functor. Use of the `inject'` operator results in a full tree traversal, which is hardly performant for an operation that is effectively just relabeling the type of subdata. Third, while we have in fact used `three` in our definition of `nine`, we will have to craft additional smart constructors to hide the mess we have made, e.g,

```
times_ x y = In (inj ((inject' x) `Times` (inject' y)))
```

5.4.2 First Class Variants

The approaches given by DTC and VAR are encodings of extensible variants into Haskell; this is to say, they use the primitives, tooling, and language extensions available in Haskell/GHC to build extensible variants. But there is a burden on the programmer in using these encodings. At a minimum, the programmer should be aware of the concretization problem, and write explicit most-general type annotations to all of their terms.

An alternative not discussed in this report to extensible variant encodings would be row types [10], where records and variants are intrinsic to the underlying type system. Row types are by now well studied and have been implemented in Haskell systems in the past. While a proper summary of row typing exceeds the scope of this report, it is worth noting that there

is a path out of encodings and into language support for the features we have discussed. A perfectly sensible take-away from this report might be to conclude that the encodings are in fact too burdensome.

6 Conclusion

This report has walked through the evolution of extensible variants in Haskell to overcome the expression problem. We began with Liang et al. to introduce the general strategy: define your cases, combine the cases with a sum type operator, and define evaluation for the summation of these cases via a typeclass. DTC generalized this approach further with functorially represented components, permitting better extensibility. Finally, VAR extended DTC's inclusion class to accomodate left-nested coproducts, and introduced a branching operator to provide case evaluation sans typeclasses.

The above covers the technical contributions of these papers, as well as technical barriers. We have yet to address the practicality of actually using these encodings. The obvious, but blunt, question is: would you use these encodings in your own work? Further, would these encodings be suitable for a sufficiently complex compiler or interpreter? This is akin to asking if the gains in modularity and generality outweigh the technical burdens imposed by the encoding itself. I struggle to answer in the affirmative, and find the problems outlined above would pose more headache than relief. In addition, certain features of the DTC and VAR encodings cannot be implemented in Haskell as it stands today. Particularly lacking is a resolution to our concretization problem, which could be addressed via a defaulting mechanism, e.g, for the default instantiation of `f` in

```
three :: (Val ∈ f, Add ∈ f) => Expr f
```

to be guessed as `Val ⊕ Add`. Such a defaulting mechanism is not exposed in GHC, but does appear in other ambiguities, such as resolving the default type of a numeric value like `2` to **`Int`** (rather than **`Double`**).

VAR concludes in part by asking how best to provide features like extensible variants in Haskell, offering the following answers: A) all of the encodings are too complex, and unlikely to be useful in practice; B) the encodings are complex because the features are complex, and the burden to a programmer is not too high; or C) we are most of the way there, and a few further tweaks might get us the rest of the way. The paper takes the third perspective, while I am inclided to agree with the first. In particular, encoding extensible variants into Haskell follows from a lack of support in the underlying type system, but the theory is there in the form of row typing. Further, Morris and McKinna [6] have demonstrated progress in this direction with Rose, a proposed language with support for first-class extensible data types. This is the direction I would hope for (and pursue) in future research.

References

[1] Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (July 2009), 107–115.

[2] Liang, S., Hudak, P., and Jones, M. P. Monad transformers and modular interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995* (1995), R. K. Cytron and P. Lee, Eds., ACM Press, pp. 333–343.

[3] Meijer, E., g Maarten M. Fokkinga, and Paterson, R. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30,*

1991, Proceedings (1991), J. Hughes, Ed., vol. 523 of *Lecture Notes in Computer Science*, Springer, pp. 124–144.

[4] Morris, J. G. Variations on variants. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, 2015), B. Lippmeier, Ed., Haskell '15, ACM, pp. 71–81.

[5] Morris, J. G., and Jones, M. P. Instance chains: Type class programming without overlapping instances. *SIGPLAN Not. 45*, 9 (sep 2010), 375–386.

[6] Morris, J. G., and McKinna, J. Abstracting extensible data types: Or, rows by any other name. *Proc. ACM Program. Lang. 3*, POPL (jan 2019).

[7] Sheard, T., and Pasalic, E. Two-level types and parameterized modules. *Journal of Functional Programming 14*, 5 (2004), 547–587.

[8] Swierstra, W. Data types à la carte. *Journal of Functional Programming 18*, 4 (July 2008), 423–436.

[9] Wadler, P. The expression problem. `https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998.

[10] Wand, M. Complete type inference for simple objects. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987* (1987), IEEE Computer Society, pp. 37–44.