# The Expression Problem

understand the **expression problem**

see **Haskell** and **Scala** code **illustrating** the **problem**

Learn how **FP typeclasses** can be used to solve the **problem**

see the **Haskell solution** to the **problem** and a **translation** into **Scala**

Part 2

based on the work of

Haskell

Scala


**Ralf Lämmel**
**@reallynotabba**

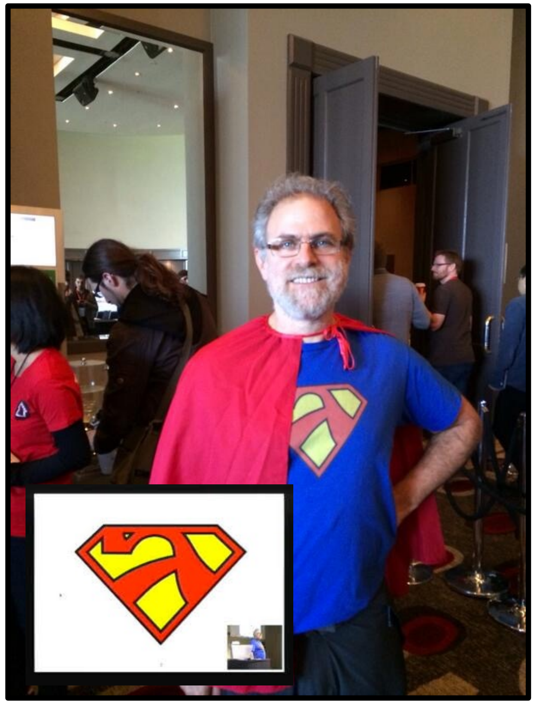


slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

@philip_schwarz

We begin Part 2 with the last slide from Part 1, which defines the **expression problem**.

Here is the definition of the **Expression Problem**.



Computer Scientist **Philip Wadler**

Cc: Philip Wadler <wadler@research.bell-labs.com>
Subject: The Expression Problem
Date: Thu, 12 Nov 1998 14:27:55 -0500
From: Philip Wadler <wadler@research.bell-labs.com>

The **Expression Problem**
Philip Wadler, 12 November 1998

The **Expression Problem** is a new name for an old problem. The goal is to define a **datatype** by **cases**, where one can **add new cases** to the **datatype** and **new functions** over the **datatype**, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take **expressions** as the **data type**, begin with one **case** (**constants**) and one **function** (**evaluators**), then add one more **construct** (**plus**) and one more **function** (**conversion to a string**).

Whether a language can solve the **Expression Problem** is a salient indicator of its capacity for expression. One can think of **cases** as **rows** and **functions** as **columns** in a table. In a **functional language**, the **rows** are fixed (**cases** in a **datatype** declaration) but it is **easy to add new columns** (**functions**). In an **object-oriented language**, the **columns** are fixed (methods in a class declaration) but it is **easy to add new rows** (**subclasses**). We want to make it **easy to add either rows or columns**.

…

https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

This deck is going to be largely based on extracts from two talks given by **Ralf Lämmel**

1. **The Expression Problem**
2. **Advanced Functional Programming – Type Classes**

The talks can be found here:

https://web.archive.org/web/20100907194522/http://channel9.msdn.com/tags/C9+Lectures/

And slides can be found here:

https://userpages.uni-koblenz.de/~laemmel/paradigms1011/resources/pdf/xproblem.pdf
https://userpages.uni-koblenz.de/~laemmel/paradigms1011/resources/pdf/typeclasses.pdf

Ralf Lämmel
@reallynotabba

The Expression Problem is an interesting software extensibility challenge.

It is interesting for us in this context because it helps us study some subtle differences between OOP and FP.

And in fact it will allow me, not today, but perhaps in the next presentation, to bring up some new supernatural powers of Haskell, because it is a real challenge, and it turns out that this challenge can be addressed with some designated Haskell expressiveness.

The Expression Problem
Ralf Lämmel
Software Language Engineer
University of Koblenz-Landau
Germany

Ralf Lämmel
@reallynotabba

Let me explain the problem first. We are at the Haskell prompt and we are entering some expressions. In fact we are playing with an expression language.

We have constant expressions.

```
> let x = Const 40
> let y = Const 2
```

We have addition expressions.

```
> let z = Add x y
```

So we can build arithmetic expressions. We use the constructors of an Algebraic Data Type (ADT). We construct terms and those terms denote arithmetic expressions.

As you see, we can pretty print those expressions.

```
> prettyPrint z
"40 + 2"
```

And we can also evaluate those expressions.

```
> evaluate z
42
```

So no big deal. The question that leads to the expression problem is:

> How can we program such an interpreter and such a pretty printer, how can we implement such an expression language, so that later on we can easily add more expression forms, such as subtraction or negation, and we can also easily add more operations, such as optimization and code generation? How can we set up our programming style so that such extensions are possible?

That is the expression problem.

Ralf Lämmel
@reallynotabba

Here is an **expression problem** summary.

- **Program** = **data** + **operations**
- There could be ***many*** **data variants**.

  e.g. expression forms: **constant**, **addition**.

- There could be ***many*** **operations**.

  e.g. **pretty printing**, **evaluation**.

- **Data** and **operations** should be ***extensible***.

The **expression problem** is whether or not, and if so how, we can **add data variants** and **operations**.

**This sounds like a simple problem, but as you will see, it is not so easy to address this problem in a satisfactory manner in functional and OO programming**.

**At least not as long as we are limiting ourselves to basic functional programming and basic OO programming**.

**Ralf Lämmel**
@reallynotabba

I should make sure that we have some shared understanding of what I mean by **extensibility**, being **extensible** in the **data dimension** and the **operation dimension**.

What I mean by that is that we should take care of at least **three requirements**:

1. **Code-level modularization**

   If you have a given program and you want to extend it, then this extension should be in a new code unit, we shoud not allow ourselves to extend the program by going back into existing code units and editing them. **This is what we mean by extensibility, that we do not touch existing code**.

2. **Separate compilation**

   We want our basic program and our extensions to be true modules in the sense of compilation and deployment. So suppose we have a program, we compile it and we ship it, it is running at the customer site. Now the customer requests an extension to the program, then we should be able to develop this extension by means of another module which we can compile in separation. **We don't have to recompile anything that was there before, and so we can deliver the extension to the customer just by shipping that new module for the extension.**

3. **Static type safety**

   Suppose we are using a language like **C#** or **Java**, with some sophisticated means of type checking to help us with avoiding certain types of programming errors, then we want to preserve that type checking power even in the view of extensibility. **Just because our program is becoming extensible, we don't want to compromise on type safety**.

**Extensibility**

Three Requirements:

1. Code-level modularization
2. Separate compilation
3. Static type safety

# Pretty printing and evaluating expressions with Haskell

1st module — data variants

```haskell
module Data where

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

2nd module — one operation

```haskell
module Evaluator where

import Data

evaluate :: Expr -> Int
evaluate (Const i) = i
evaluate (Add l r) = evaluate l + evaluate r
```

3rd module — another operation

```haskell
module PrettyPrinter where

import Data

prettyPrint :: Expr -> String
prettyPrint (Const i) = show i
prettyPrint (Add l r) = prettyPrint l ++ " + " ++ prettyPrint r
```

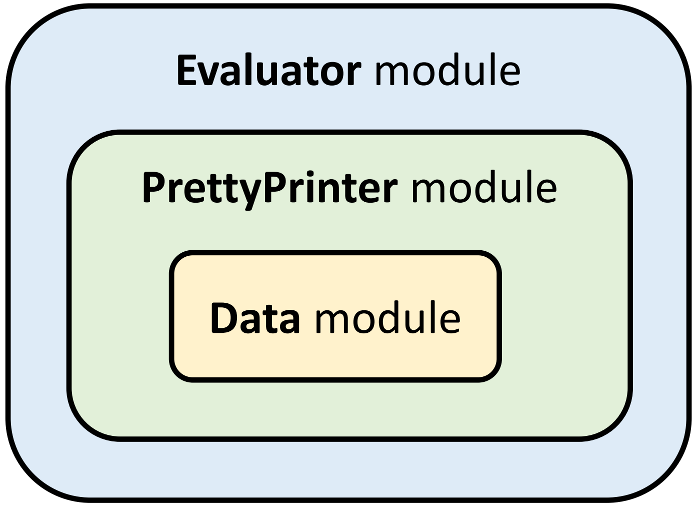
**Ralf Lämmel**
@reallynotabba

So we have this **chain of extensions**, if you like.

We start with the **Data module**.

On top of that **first module**, we define a **second module** for the **prettyPrint operation**.

And on top of that **second module** we define a **third module** for the **evaluation operation**.



On the next slide we translate that **Haskell** program into **Scala**.

**@philip_schwarz**

```haskell
module Data where

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

```haskell
module Evaluator where

import Data

evaluate :: Expr -> Int
evaluate (Const i) = i
evaluate (Add l r) = evaluate l + evaluate r
```

```haskell
module PrettyPrinter where

import Data

prettyPrint :: Expr -> String
prettyPrint (Const i) = show i
prettyPrint (Add l r) = "(" ++ prettyPrint l ++ " + " ++
                         prettyPrint r ++ ")"
```

```haskell
main :: IO ()
main = let expression = (Add (Const 2) (Add (Const 3) (Const 4)))
  in do print (prettyPrint expression)
        print (evaluate expression)
```

```
haskell> main
"(2 + (3 + 4))"
9
```

```scala
object Data:

  enum Expr:
    case Const(i: Int)
    case Add(l: Expr, r: Expr)
```

```scala
import Data.Expr, Expr._

object Evaluator:

  def evaluate(expr: Expr): Int = expr match
    case Const(i) => i
    case Add(l,r) => evaluate(l) + evaluate(r)
```

```scala
import Data.Expr, Expr._

object PrettyPrinter:

  def prettyPrint(expr: Expr): String = expr match
    case Const(i) => i.toString
    case Add(l,r) => "(" ++ prettyPrint(l) ++ " + " ++
                      prettyPrint(r) ++ ")"
```

```scala
@main def main: Unit =
  val expression: Expr = Add(Const(2),Add(Const(3),Const(4)))
  println(prettyPrint(expression))
  println(evaluate(expression))
```

```
scala> main
(2 + (3 + 4))
9
```

The situation that we have with basic functional programming is that it is easy to carry out operation extensions, as we have just demonstrated, it is easy to perform new functions on existing data. It is however not easy to perform data extensions.

So why is that, and what do I mean by data extension?

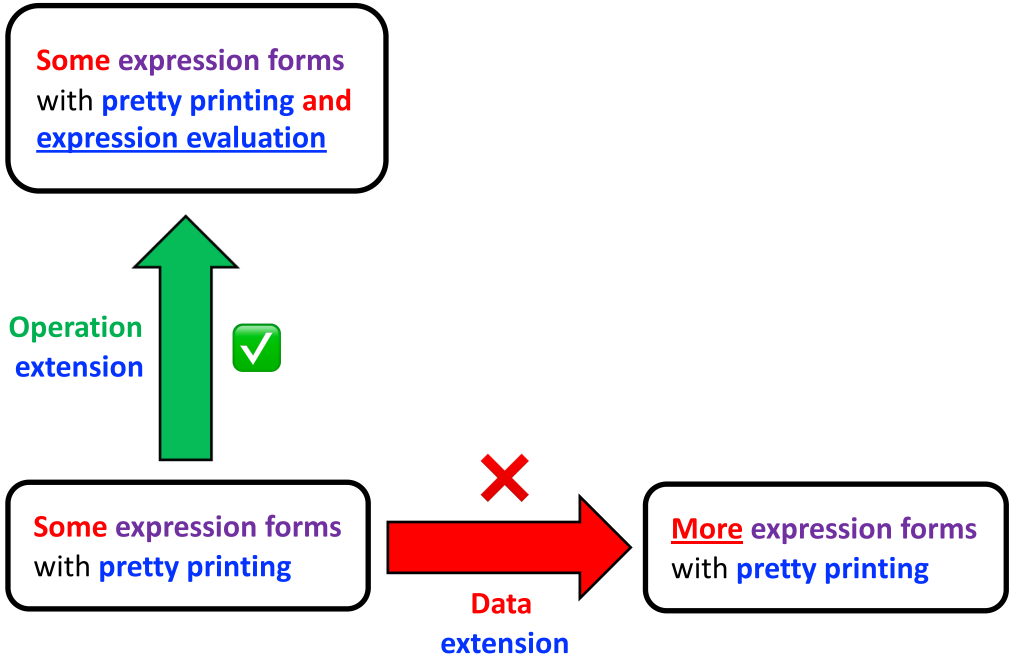
Well, by data extension I mean of course that we could add another data variant, another expression form, without touching existing code, and we can also make sure that all the existing operations, like in our example, let's say, pretty printing, works for this new expression form.

Why is this not easy, or even possible in basic functional programming?

It is because algebraic data types are closed, and in fact, also recursive function definitions, defined by pattern matching are closed too, in basic functional programming, and because they are closed, there is no way, in a module that is laid on top of the basic data variants, to add data variants, there is just no way to do this.

Again, we could of course go back to the data module and patch it, but this is not extensibility because we would touch existing code units.

So this is interesting: with functional programming we only get operation extensions easily, but we don't get data extensions easily.




It is easy to add operations in basic functional programming.

It is not so easy to add data variants (without touching existing code).

Ralf Lämmel

@reallynotabba

In the next part of his talk on the **Expression Problem**, **Ralf Lämmel** uses **C#** as an **OOP** language.

Instead of showing his **C#** code examples, we'll show the equivalent **Scala** code (the language supports both **OOP** and **FP**).

Now let's try the same experiment with Scala.

In Scala you might think we could start from these classes here, so these classes would more or less resemble the algebraic data type of our Haskell development…

You might think that this is a good initial program: we compile it, we ship it, the customer uses it, and now the customer says: hey, this is a great program, but I would like to have an evaluator for this program, and then we say OK, no problem, we just have to add an evaluate method to those classes.

Well, that's where we are in trouble, because how do we do this? We have to violate separate compilation, right? In order to supply an extension for evaluation we would need to touch this code and add another method to it, we would have to recompile and ship those classes again to the customer, and he would need to throw away those existing classes and install a new version.

This is bad extensibility. But remember, this was easy with Haskell: we could easily add prettyPrinting and subsequently evaluation, so with OOP we fail to do operation extensions, even though they were easy with Haskell.

**Ralf Lämmel**
@reallynotabba

```scala
trait Expr:
  def prettyPrint: String
```

```scala
case class Const(i: Int) extends Expr:
  def prettyPrint: String = i.toString
```

```scala
case class Add(l: Expr, r: Expr) extends Expr:
  def prettyPrint: String = "(" + l.prettyPrint + " + " +
                                  r.prettyPrint + ")"
```

```scala
@main def main: Unit =
  val expr: Expr = Add(Const(2),Add(Const(3),Const(4)))
  println(expr.prettyPrint)
```

```
scala> main
(2 + (3 + 4))
```

OK, but **we can do something with OOP here that we couldn't do with FP, we can do data extensions easily.**

We can of course always go and add another class, in this case we add a class **Neg** for negation, with one operand for which negation is to be computed, and we add an implementation to the initial system which already has a few **expression forms**, and which also has snapshotted the **prettyPrint operation** in those classes, so because **Expr** has a **prettyPrint operation**, our implementation **Neg** also has a **prettyPrint operation**, no surprise.

**This is a data extension**. It is more than just the **data structure**, it also defines the case for all preexisting **operations**, in our case we only have one **operation**, **prettyPrint**.

So this is interesting, right? **We can perform data extensions**, as you have just seen. We had this initial program, with some **data variants** and some **operations**, in this case **pretty printing**, and **we can go and add one data extension, perhaps another data extension, and so on.**

**Ralf Lämmel**
@reallynotabba

```scala
trait Expr:
  def prettyPrint: String
```

```scala
case class Const(i: Int) extends Expr:
  def prettyPrint: String = i.toString
```

```scala
case class Add(l: Expr, r: Expr) extends Expr:
  def prettyPrint: String = "(" + l.prettyPrint + " + " +
                            r.prettyPrint + ")"
```

```scala
@main def main: Unit =
  val expr: Expr = Add(Const(2),Add(Const(3),Const(4)))
  println(expr.prettyPrint)
  println(Neg(expr).prettyPrint)
```

```scala
case class Neg(expr: Expr) extends Expr:
  def prettyPrint: String = "-" + expr. prettyPrint
```

```
scala> main
(2 + (3 + 4))
-(2 + (3 + 4))
```

Data extension for negation

Initial data variants with pretty printing

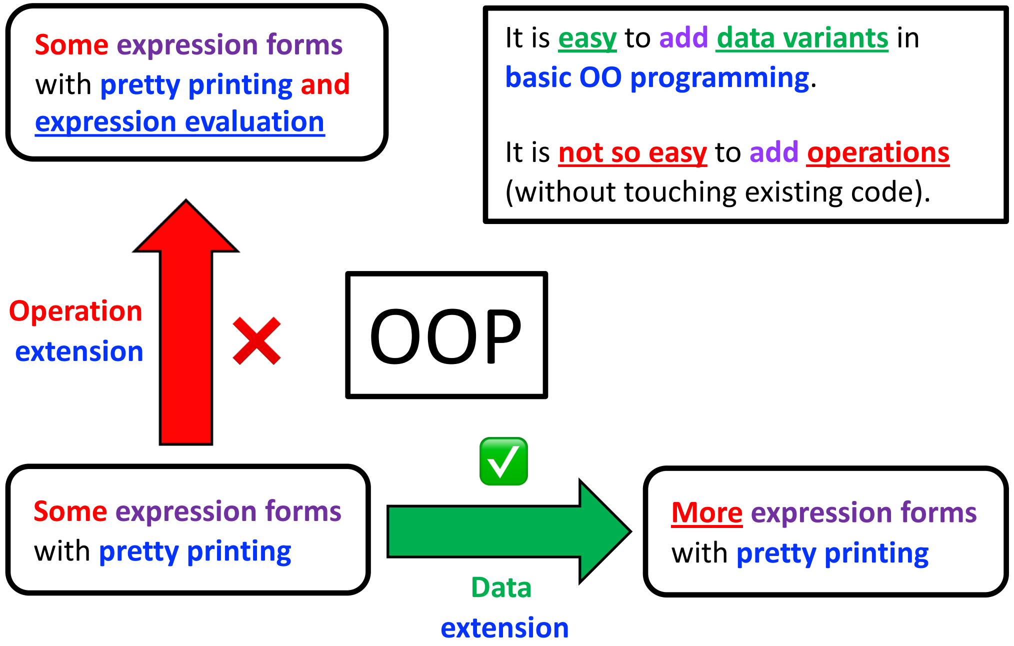
Ralf Lämmel
@reallynotabba

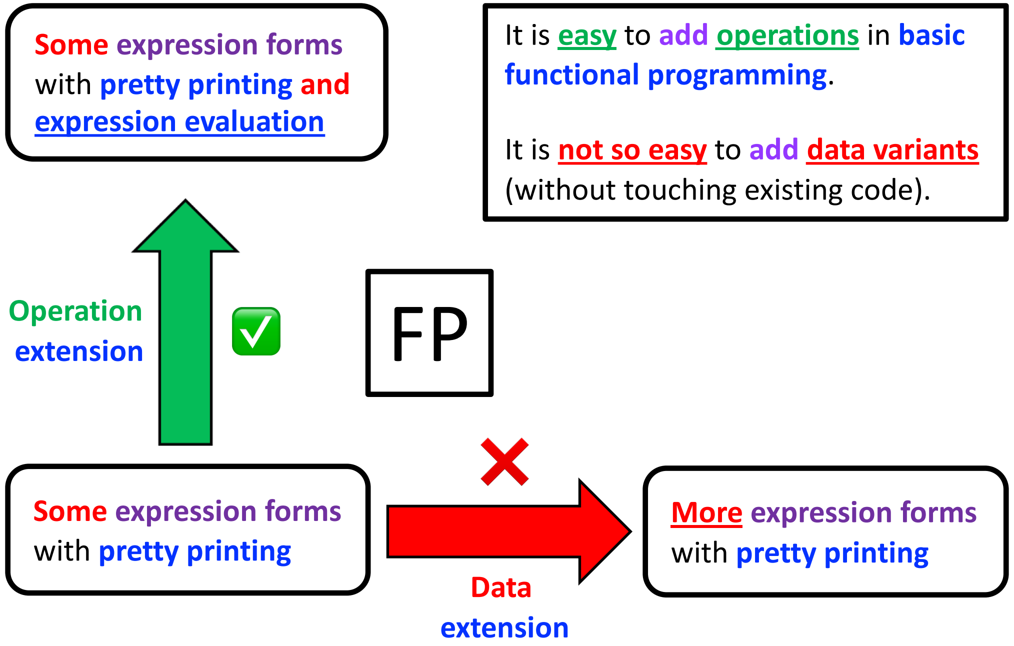
This is interesting, also because it means the situation is pretty much the inverse compared to FP.

So we can do data extension. We can go from a program that covers some expression forms to a program that covers more expression forms. We couldn't do that with Haskell. However, we can't do operation extension in OOP, because we are not supposed to add methods to existing classes without violating separate compilation.

So it seems like FP and OOP are complementary, which is interesting.

There are two subtle things worth pointing out. You should realise that I quite often say basic OOP and basic FP. And you should also realise that I say '(not so) easy to add'. By basic I mean what you learn in a 101 OOP/FP course. If you go nuts and use every weapon available you can also get operation extensibility in OOP. And then when I say it is not so easy to add operations, this is part of the same story: if you are willing to engage in sophisticated encodings, well then you can get both dimensions of extensibiity, but the point is that you don't want to do crazy things, you want to use relatively straightforward idioms and design patterns, and still like to get both dimensions of extensibility.



Remember this table from Part 1?

| | | Addition of new | |
|---|---|---|---|
| | | Function | Type |
| Polymorphism | Subtype | OCP✗ | OCP✓ |
| | Alternation-based ad-hoc | OCP✓ | OCP✗ |

Here is an updated version that uses the same terminology seen in the two diagrams on the previous slide.

| | Operation Extension | Data Extension |
|---|---|---|
| OOP | ✗ | ✓ |
| FP | ✓ | ✗ |

**Ralf Lämmel**
@reallynotabba

## Summary

How are we supposed to design a program so that we can achieve both **data extensibility** and **operation extensibility**?

What **language concepts** help us achieve **both dimensions of extensibility** (and **separate compilation** and **static type safety**)?

In the presentation called **The Expression Problem**, **Ralf Lämmel** does not cover the **solution** to the **problem**.

The presentation in which he does that is called **Advanced Functional Programming – Type Classes**.

Ralf Lämmel
@reallynotabba

Remember again what it means to solve the expression problem.

It means that we can do data extensions and operation extensions, and we convinced ourselves that operation extensions are straightforward in Haskell, or any FP language, because it is easy to define new functions in an FP language.

The hard part is to do data extension in an FP language. Data extensibility is difficult because the standard algebraic data types of Haskell and other languages in the functional paradigm are closed.

We need to open up data types.

We need some encoding scheme to get open data types, and this is what type classes will provide us with.

And then remember, it is not enough just to be able to have new data variants, to have open data types, no, we also need to open up functions, because the functions, whenever there is a new data variant, the existing functions also need to pick up this new data variant.

We need open data types and open functions.

Let's solve the expression problem with open data types and open functions.

Ralf Lämmel
@reallynotabba

Let’s start from the closed situation.

There are two constructors and one of them, Add, is recursive.

This is the function for which we want to achieve data extensibility, but this is of course, to start with, the closed version of it.

There is one equation per datatype constructor, and there are recursive function applications.

Point of reference:
the closed datatype

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

Point of reference:
the closed function

```
evaluate :: Expr -> Int
evaluate (Const i) = i
evaluate (Add l r) = evaluate l + evaluate r
```

So this is the reference implementation, except, it is not extensible with regard to data variants.

We need to open it up.

Ralf Lämmel

@reallynotabba

Here is how **typeclasses** help us to come up with **open data types**.

The **open datatype**

```
data Const     = Const Int
data Add l r = Add l r
```

```
class Expr x
instance Expr Const
instance (Expr l, Expr r) => Expr (Add l r)
```

This is a certain **scheme**, so let me explain this **scheme** in detail.

Ralf Lämmel
@reallynotabba

What we do here is we start from the closed data type

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

and we take its constructors, and we define one data type for each constructor.

```
data Const   = Const Int
data Add l r = Add l r
```

So now you see there are two distinct data types, one for each of the original constructors.

And then the second part of the encoding scheme is to use a typeclass to model the original data type. We had an algebraic data type (ADT) Expr. Now, to make it open, we replace the ADT Expr with typeclass Expr.

```
class Expr x
instance Expr Const
instance (Expr l, Expr r) => Expr (Add l r)
```

And then we have to say what types are expression types, and there are these types here, Const types and Add types. We have two instance types, one instance for each original constructor. And then because Add types are recursive again, we need to add the appropriate constraints here so that we say, if you form an Add type from two other subexpression types l and r, please make sure that the subexpression types are also Expr types. This is what the constraint says.

This is the scheme to define an open data type. What is remarkable about this definition is that there are no typeclass members involved and that's because we only want to model the data type, we don't yet want to model any operation, we don't want to anticipate any operations here. That will be the next step, to define operations on top of this data type. This is only the data type.

Ralf Lämmel
@reallynotabba

Here is the beginning of the open function for evaluation. It is going to be an open function, so we can't expect to see a regular function, rather, we use a function that is a typeclass member, so we designate a typeclass Evaluate to the evaluate function.

The open function
(type-class declaration)

```
class Expr x => Evaluate x

  where

    evaluate :: x -> Int
```

Point of reference:
the closed function

```
evaluate :: Expr -> Int
evaluate (Const i) = i
evaluate (Add l r) = evaluate l + evaluate r
```

Were we now have x in `evaluate :: x -> Int`, we previously had Expr, the closed algebraic data type Expr. Now, we are polymorphic in the type x here, but we constrain the type to be an Expr type.

So this is how we go from the closed function signature to the open function signature.

Ralf Lämmel
@reallynotabba

Now here is the rest of it. Here are the instances for the **Evaluate typeclass**. Obviously there are two **typeclass instances** because we have two **expression forms**

The **open function**
(type-class instances)

```
instance Evaluate Const

  where

    evaluate (Const i) = i

instance (Evaluate l, Evaluate r) => Evaluate (Add l r)

  where

    evaluate (Add l r) =

      evaluate l + evaluate r
```

The **open function**
(**typeclass** declaration)

```
class Expr x => Evaluate x

  where

    evaluate :: x -> Int
```

Point of reference:
the **closed function**

```
evaluate :: Expr -> Int
evaluate (Const i) = i
evaluate (Add l r) = evaluate l + evaluate r
```

These are exactly the definitions as we had them before in the **closed** model, where we had **equations**, the only difference is that these definitions here are not just the plain list of **equations**, but rather, they are **integrated** into this **system of instances**, and these **instances** make sure that these **definitions** apply to the appropriate **expression forms**.

Because we recurse on the **subexpressions** of **Add**, we have to make sure that the types of the **subexpressions** are such that we can perform **Evaluation**, so again we have a **constraint** in the **instance** for **Add**.

**Ralf Lämmel**
@reallynotabba

**We have solved the Expression Problem**. We have **open data types**, we have **open functions**, and so we can define any number of **open functions, so we have solved the problem**.

But let's just illustrate it, that we can indeed perform **data extensions** in such a setup.

It is very easy. It is three steps:

1. Declare a designated **datatype** for the **data variant**
2. **Instantiate** the **typeclass** for the open **datatype**
3. **Instantiate** all **typeclasses** for existing **operations**

a **data extension**

```
① data Expr x => Neg x = Neg x

② instance Expr x => Expr (Neg x)

③ instance Evaluate x => Evaluate (Neg x)

     where

       evaluate (Neg x) = 0 - evaluate x
```

**Expression Problem Summary**

How are we supposed to design a program so that we can achieve both **data extensibility** and **operation extensibility**?

What **language concepts** help us achieve **both dimensions of extensibility** (and **separate compilation** and **static type safety**)?

Let's solve the **expression problem** with **open data types** and **open functions**.

| | Operation Extension | Data Extension |
|---|---|---|
| OOP | ✗ | ✓ |
| FP | ✓ | ✓ |

① We first need to come up with a **new data type**, whenever there is a **new data variant**. We want to have negation, so we define a **new data type** with the **constructor Neg**, and we **constrain** it so that it is an **Expr** type.

② Then we register this **data type** with the **typeclass** for **expression forms**, we say yes, negation is indeed an **expression type**.

③ And then we say well, let's see what **operations** are around, well we have an **operation Evaluate** and so we instantiate **Evaluate** for negation, and we just implement the **evaluation** for negation as we would do in the **closed** model, there is nothing special.

@philip_schwarz

By the way, when I tried to compile that code, I got an error that suggested enabling the -**XDatatypeContexts** feature, but then I got this:

```
Main.hs:2:14: warning:
    -XDatatypeContexts is deprecated: It was widely considered a misfeature, and has been removed  from the Haskell language.
  |
2 | {-# LANGUAGE DatatypeContexts #-}
  |              ^^^^^^^^^^^^^^^^
```

So in the code, I replaced

```haskell
data Expr x => Neg x = Neg x
```

with

```haskell
data Neg x = Neg x
```

The next slide shows the whole of the **Haskell** code solving the **expression problem**, plus a **prettyPrint** **function**.

It also shows how the **solution** code supports both **operation extension** and **data extension**.

**Data Extension**

Adding a new **expression form** without modifying existing code

**Operation Extension**

Adding a new **function** without modifying existing code

```
data Const = Const Int
data Add l r = Add l r
```

```
class Expr x
```

```
instance Expr Const
instance (Expr l, Expr r) => Expr (Add l r)
```

```
class Expr x => Evaluate x
  where evaluate :: x -> Int
```

```
instance Evaluate Const
  where evaluate (Const i) = i

instance (Evaluate l, Evaluate r) =>
  Evaluate (Add l r)
    where evaluate (Add l r) =
      evaluate l + evaluate r
```

(1) first **operation extension**: adding the **evaluate function**

```
data Neg x = Neg x
```

```
instance Expr x => Expr (Neg x)

instance Evaluate x => Evaluate (Neg x)
  where evaluate (Neg x) = 0 - evaluate x
```

(2) **data extension** – adding the **negation expression**

```
class Expr x => PrettyPrinter x
  where prettyPrint :: x -> String
```

```
instance PrettyPrinter Const
  where prettyPrint (Const i) = prettyPrint i

instance (PrettyPrinter l, PrettyPrinter r) => PrettyPrinter (Add l r)
  where prettyPrint (Add l r) =
    "(" ++ (prettyPrint l) ++ "+" ++ (prettyPrint r) ++ ")"

instance PrettyPrinter x => PrettyPrinter (Neg x)
  where prettyPrint (Neg x) = "-" ++ (prettyPrint x)
```

(3) second **operation extension**: adding the **prettyPrint function**

Let’s take that **Haskell** code for a quick spin.

```
four = Const 4
twoPlusThree = Add (Const 2) (Const 3)
twoPlusThreeNegated = Neg twoPlusThree

main :: IO ()
main = do

  putStrLn (show (evaluate four))
  putStrLn (show (evaluate twoPlusThree))
  putStrLn (show (evaluate twoPlusThreeNegated))

  putStrLn (show (prettyPrint four))
  putStrLn (show (prettyPrint twoPlusThree))
  putStrLn (show (prettyPrint twoPlusThreeNegated))
```

```
haskell> main
4
5
-5
"4"
"(2+3)"
"-(2+3)"
```

Next, I got started having a go at a **Scala** translation of the **Haskell** code, but I soon got stuck, so I asked for suggestions in the **Scala** users forum.

Scala USERS

## Translating Haskell Expression Problem Solution to Scala 3

Question

**philipschwarz** 29d

Hi all,

Having worked on The Expression Problem - Part 1 (1) (download for proper image quality), I am now starting out on Part 2, and I am asking myself if it is possible to translate the following Haskell solution of the Expression Problem into Scala 3.

I received suggestions from both **Alex Boisvert** and **Michael Marte** (Thank you both very much). The next slide is the **Scala** equivalent of the previous slide, and consists of my very minor tweaks and additions to **Michael**'s translation.

**Michael Marte**
informarte

https://users.scala-lang.org/t/translating-haskell-expression-problem-solution-to-scala-3/8303/3

Btw, in order to fit the code onto the slide, I called the **pretty printer** typeclass **Show**.

@philip_schwarz

```scala
case class Const(c: Int)
case class Add[A, B](l: A, r: B)
```

```scala
trait Expr[A]
```

```scala
given Expr[Const] with { }
given [A, B](using leftExpr: Expr[A], rightExpr: Expr[B]): Expr[Add[A, B]] with { }
```

**Data Extension**

Adding a new **expression form** without modifying existing code

**Operation Extension**

Adding a new **function** without modifying existing code

```scala
trait Eval[A]:
  def eval(a: A)(using expr: Expr[A]): Int
```

```scala
given Eval[Const] with
  def eval(a: Const)(using expr: Expr[Const]) = a.c

given [A, B](using leftExpr: Expr[A], rightExpr: Expr[B],
                   leftEval: Eval[A], rightEval: Eval[B]): Eval[Add[A, B]] with
  def eval(a: Add[A, B])(using expr: Expr[Add[A, B]]) =
    leftEval.eval(a.l) + rightEval.eval(a.r)
```

① first **operation extension**: adding the **evaluate function**

```scala
case class Neg[A](a: A)
```

```scala
given [A](using expr: Expr[A]): Expr[Neg[A]] with { }

given [A](using expr: Expr[A], subEval: Eval[A]): Eval[Neg[A]] with
  def eval(a: Neg[A])(using expr: Expr[Neg[A]]) = -subEval.eval(a.a)
```

② **data extension** – adding the **negation expression**

```scala
trait Show[A]:
  def show(a: A)(using expr: Expr[A]): String
```

```scala
given Show[Const] with
  def show(a: Const)(using expr: Expr[Const]) = a.c.toString

given [A, B](using leftExpr: Expr[A], rightExpr: Expr[B],
                   leftShow: Show[A], rightShow: Show[B]): Show[Add[A, B]] with
  def show(a: Add[A, B])(using expr: Expr[Add[A, B]]) =
    "(" ++ leftShow.show(a.l) ++ "+" ++ rightShow.show(a.r) ++ ")"

given [A](using expr: Expr[A], subShow: Show[A]): Show[Neg[A]] with
  def show(a: Neg[A])(using expr: Expr[Neg[A]]) = "-" ++ subShow.show(a.a)
```

③ second **operation extension**: adding the **prettyPrint function**


```
def eval[A:Expr:Eval](a: A) = implicitly[Eval[A]].eval(a)
def show[A:Expr:Show](a: A) = implicitly[Show[A]].show(a)

val four = Const(4)
val twoPlusThree = Add(Const(2), Const(3))
val twoPlusThreeNegated = Neg(twoPlusThree)

@main def main: Unit =

  println(eval(four))
  println(eval(twoPlusThree))
  println(eval(twoPlusThreeNegated))

  println(show(four))
  println(show(twoPlusThree))
  println(show(twoPlusThreeNegated))
```

```
scala> main
4
5
-5
4
(2+3)
-(2+3)
```

The next slide shows the **Haskell** and **Scala** code side by side.

Again, to save space, the **pretty printer typeclass** is now called **Show**.

```haskell
data Const = Const Int
data Add l r = Add l r
```

```haskell
class Expr x
```

```haskell
instance Expr Const
instance (Expr l, Expr r) => Expr (Add l r)
```

```haskell
class Expr x => Evaluate x
  where evaluate :: x -> Int
```

```haskell
instance Evaluate Const
  where evaluate (Const i) = i

instance (Evaluate l, Evaluate r) =>
  Evaluate (Add l r)
    where evaluate (Add l r) =
      evaluate l + evaluate r
```

```haskell
data Neg x = Neg x
```

```haskell
instance Expr x => Expr (Neg x)

instance Evaluate x => Evaluate (Neg x)
  where evaluate (Neg x) = 0 - evaluate x
```

```haskell
class Expr x => Show x
  where show :: x -> String
```

```haskell
instance Show Const
  where show (Const i) = show i

instance (Show l, Show r) => Show (Add l r)
  where show (Add l r) =
    "(" ++ (show l) ++ "+" ++ (show r) ++ ")"

instance Show x => Show (Neg x)
  where show (Neg x) = "-" ++ (show x)
```

```scala
case class Const(c: Int)
case class Add[A, B](l: A, r: B)
```

```scala
trait Expr[A]
```

```scala
given Expr[Const] with { }
given [A, B](using leftExpr: Expr[A], rightExpr: Expr[B]): Expr[Add[A, B]] with { }
```

```scala
trait Eval[A]:
  def eval(a: A)(using expr: Expr[A]): Int
```

```scala
given Eval[Const] with
  def eval(a: Const)(using expr: Expr[Const]) = a.c

given [A, B](using leftExpr: Expr[A], rightExpr: Expr[B],
                   leftEval: Eval[A], rightEval: Eval[B]): Eval[Add[A, B]] with
  def eval(a: Add[A, B])(using expr: Expr[Add[A, B]]) =
    leftEval.eval(a.l) + rightEval.eval(a.r)
```

```scala
case class Neg[A](a: A)
```

```scala
given [A](using expr: Expr[A]): Expr[Neg[A]] with { }

given [A](using expr: Expr[A], subEval: Eval[A]): Eval[Neg[A]] with
  def eval(a: Neg[A])(using expr: Expr[Neg[A]]) = -subEval.eval(a.a)
```

```scala
trait Show[A]:
  def show(a: A)(using expr: Expr[A]): String
```

```scala
given Show[Const] with
  def show(a: Const)(using expr: Expr[Const]) = a.c.toString

given [A, B](using leftExpr: Expr[A], rightExpr: Expr[B],
                   leftShow: Show[A], rightShow: Show[B]): Show[Add[A, B]] with
  def show(a: Add[A, B])(using expr: Expr[Add[A, B]]) =
    "(" ++ leftShow.show(a.l) ++ "+" ++ rightShow.show(a.r) ++ ")"

given [A](using expr: Expr[A], subShow: Show[A]): Show[Neg[A]] with
  def show(a: Neg[A])(using expr: Expr[Neg[A]]) = "-" ++ subShow.show(a.a)
```

That’s all. I hope you found it useful.

@philip_schwarz