# The Expression Problem

Ralf Lämmel
Software Language Engineer
University of Koblenz-Landau
Germany

© 2010 Ralf Lämmel

# Demo of an **expression <u>language</u>** susceptible to the **expression <u>problem</u>**

```
> let x = Const 40
> let y = Const 2
> let z = Add x y
> prettyPrint z
"40 + 2"
> evaluate z
42
```

© 2010 Ralf Lämmel

# The Expression Problem

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

  E.g.: <u>expression</u> forms: constant, addition.
- There could be *many* operations.

  They dispatch on and recurse into data.

  E.g.: pretty printing, evaluation.
- Data & operations should be *extensible*!

© 2010 Ralf Lämmel

# Extensibility

Code-level modularization

Separate compilation

Static type safety

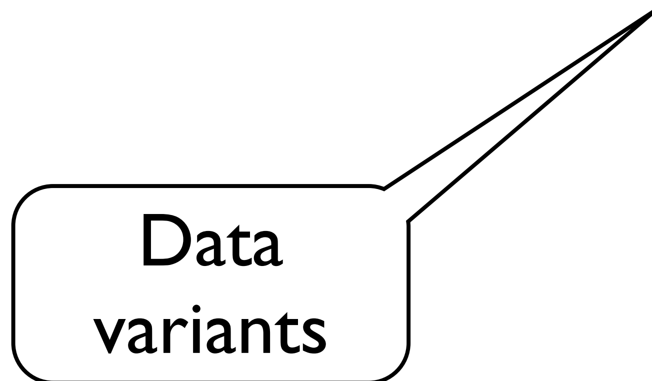
© 2010 Ralf Lämmel

# Pretty printing and evaluating expressions with Haskell

© 2010 Ralf Lämmel

```haskell
module Data where

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



© 2010 Ralf Lämmel

```haskell
module PrettyPrinter where

import Data

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

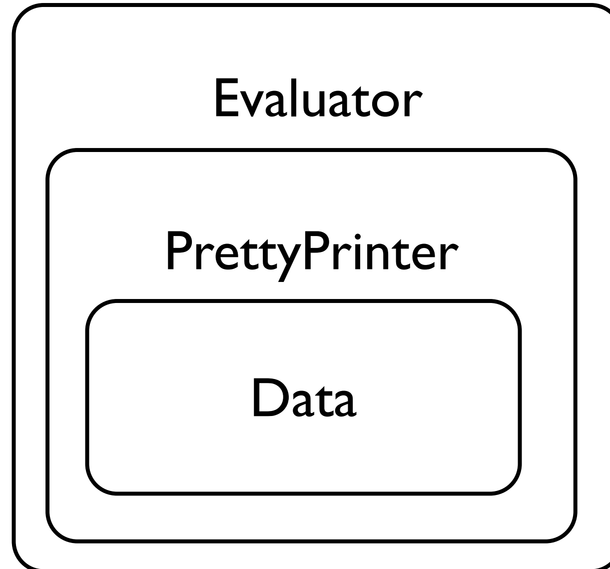
© 2010 Ralf Lämmel

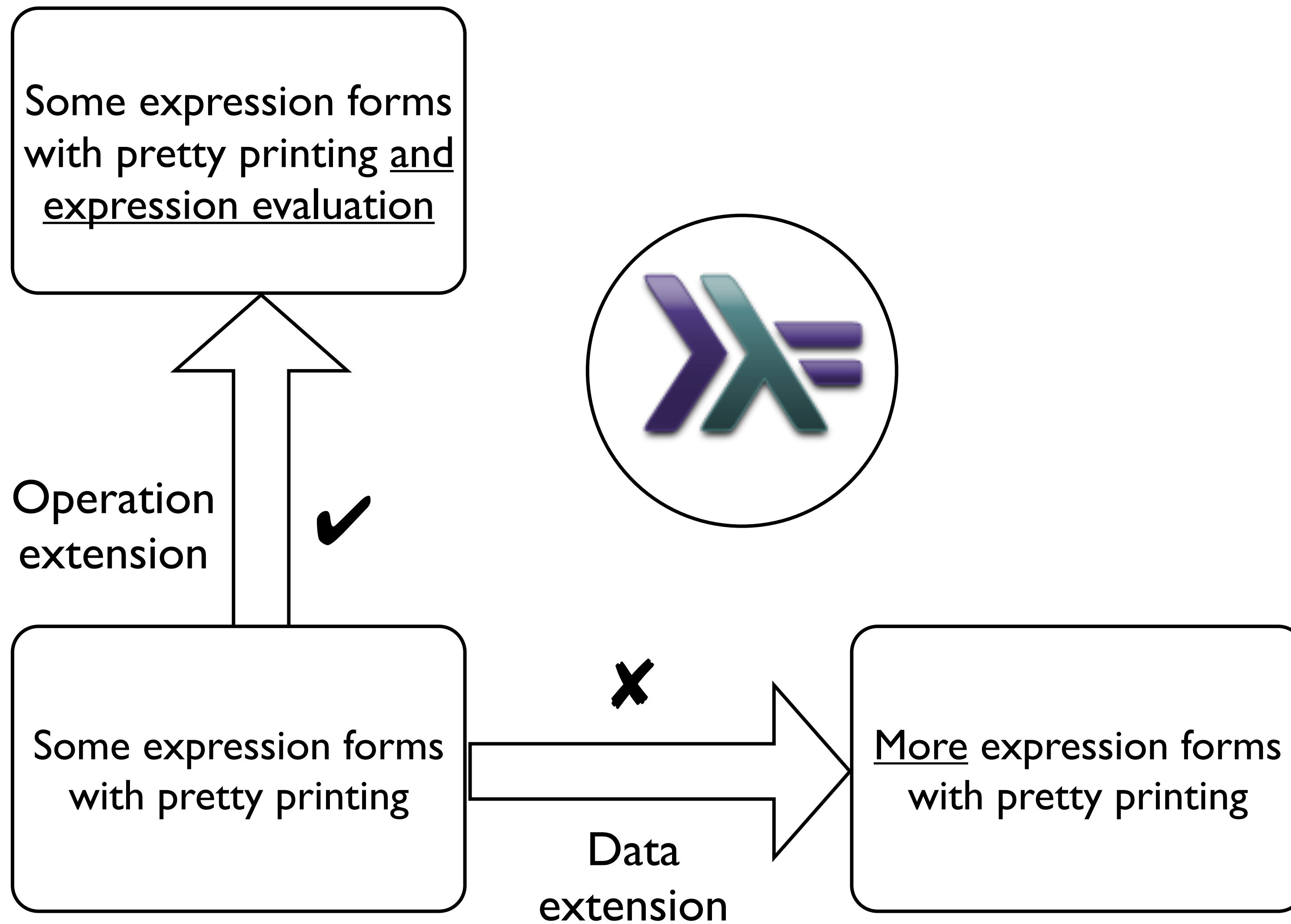
```
module Evaluator where

import Data

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

© 2010 Ralf Lämmel




Some expression forms
with pretty printing and
expression evaluation
Operation
extension
✔
Some expression forms
with pretty printing
✘
Data
extension
More expression forms
with pretty printing

© 2010 Ralf Lämmel

It’s easy to add operations in basic functional programming; it’s not so easy to add data variants (without touching existing code).

© 2010 Ralf Lämmel

# Pretty printing and evaluating expressions with C#

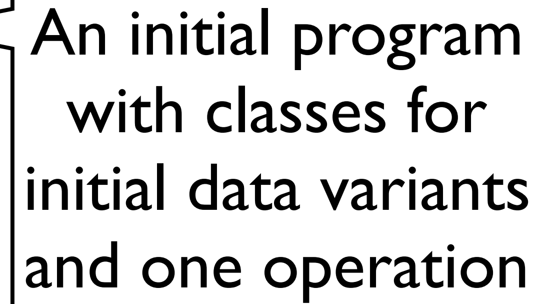
© 2010 Ralf Lämmel

The initial data variants **without any operations**

```
public abstract class Expr
{
}
public class Const : Expr
{
   public int info;
}
public class Add : Expr
{
   public Expr left, right;
}
```

© 2010 Ralf Lämmel

```
public abstract class Expr
{
   public abstract string PrettyPrint();
}
public class Const : Expr
{
   public int info;
   public override string PrettyPrint()
   {
      return info.ToString();
   }
}
public class Add : Expr
{
   public Expr left, right;
   public override string PrettyPrint()
   {
      return left.PrettyPrint() + " + " + right.PrettyPrint();
   }
}
```



© 2010 Ralf Lämmel

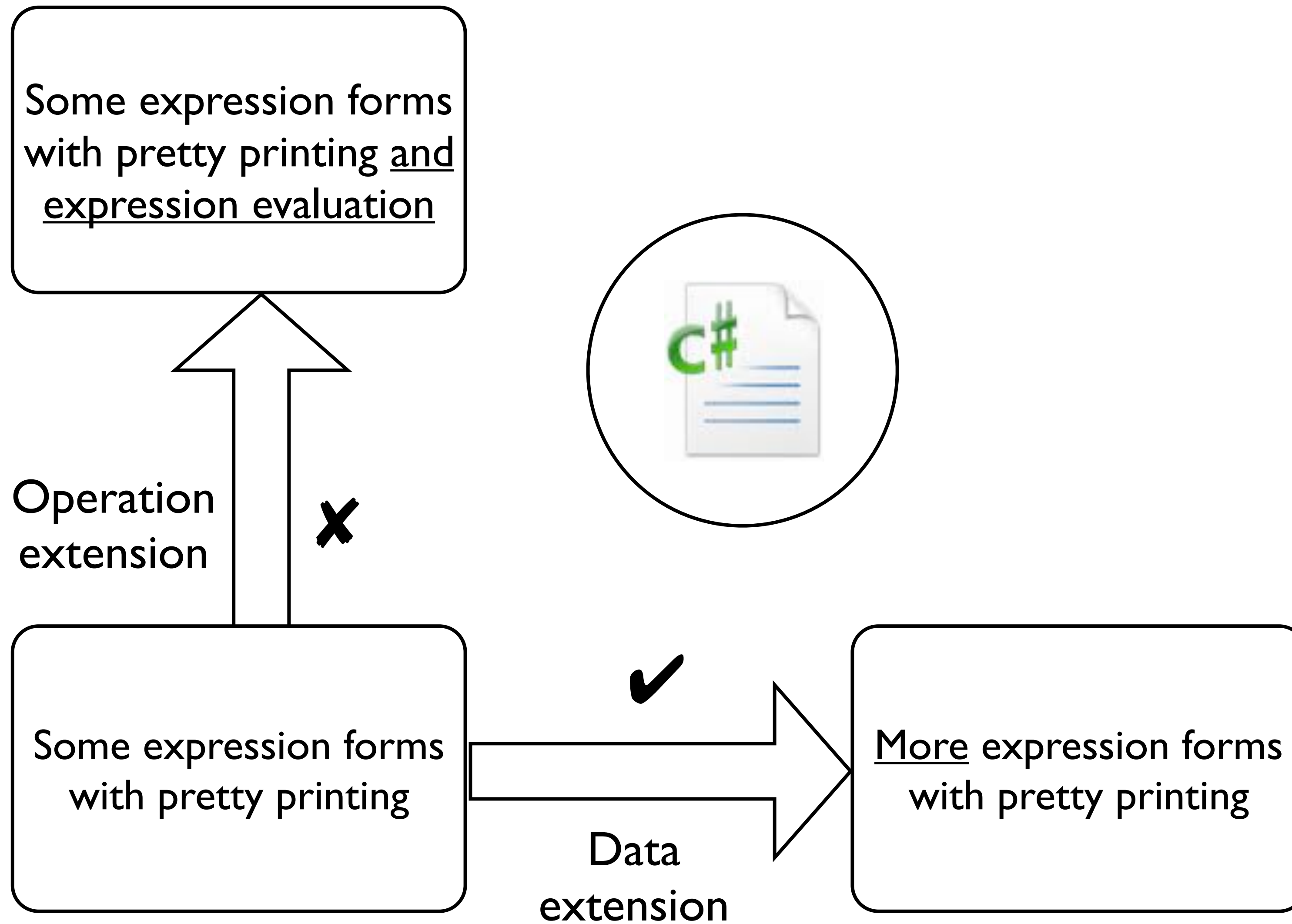
```
public class Neg : Expr
{
    public Expr operand;
    public override string PrettyPrint()
    {
        return "- (" + operand.PrettyPrint() +")";
    }
}
```

© 2010 Ralf Lämmel

Data extension for negation

Initial data variants with pretty printing

© 2010 Ralf Lämmel



© 2010 Ralf Lämmel

It's easy to add data variants in basic OO programming; it's not so easy to add operations (without touching existing code).

© 2010 Ralf Lämmel

# Extensibility

Code-level modularization

Separate compilation

Static type safety

© 2010 Ralf Lämmel

# Non-solutions in C#

- **The Visitor Pattern**

  *We get extensibility like in basic Haskell.*

- **Partial classes**

  *Let's pretend we want separate compilation!*

- **Cast-based type switch**

  *Let's pretend we want static type safety!*

- **Extension methods**

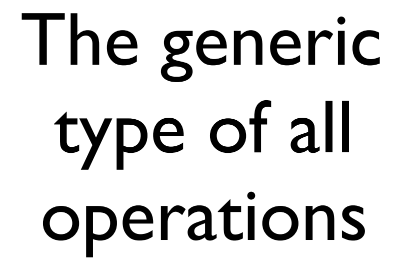
  *We need virtual methods for extensibility!*

© 2010 Ralf Lämmel

# Non-solution in C#: The Visitor Pattern

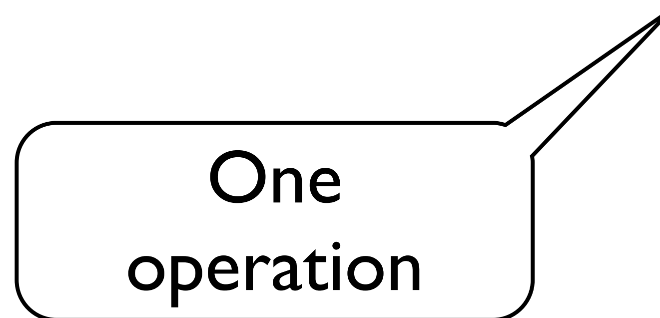
© 2010 Ralf Lämmel

```
public interface Visitor<R>
{
   R Visit(Const that);
   R Visit(Add that);
}
```



© 2010 Ralf Lämmel

```
public class PrettyPrinter : Visitor<string>
{
   public string Visit(Const that)
   {
      return that.info.ToString();
   }
   public string Visit(Add that)
   {
      return  that.left.Accept(this)
            +  " + "
            +  that.right.Accept(this);
   }
}
```



© 2010 Ralf Lämmel

```
public class Evaluator : Visitor<int>
{
    public int Visit(Const that)
    {
        return that.info;
    }
    public int Visit(Add that)
    {
        return  that.left.Accept(this)
             +  that.right.Accept(this);
    }
}
```

© 2010 Ralf Lämmel

```
public abstract class Expr
{
   public abstract R Accept<R>(Visitor<R> v);
}
public class Const : Expr
{
   public int info;
   public override R Accept<R>(Visitor<R> v)
   {
      return v.Visit(this);
   }
}
public class Add : Expr
{
   public Expr left, right;
   public override R Accept<R>(Visitor<R> v)
   {
      return v.Visit(this);
   }
}
```

© 2010 Ralf Lämmel

# A riddle with visitors

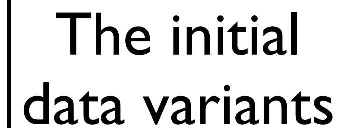
Can we extend visitors
to incorporate new
data variants?

© 2010 Ralf Lämmel

# Non-solution in C#: Partial classes

© 2010 Ralf Lämmel

```
public abstract partial class Expr
{
}
public partial class Const : Expr
{
   public int info;
}
public partial class Add : Expr
{
   public Expr left, right;
}
```



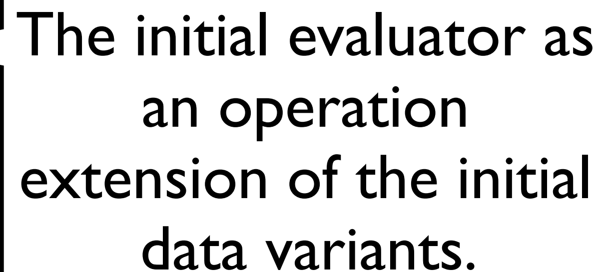
© 2010 Ralf Lämmel

```
public abstract partial class Expr
{
   public abstract string PrettyPrint();
}
public partial class Const : Expr
{
   public override string PrettyPrint()
   {
      return info.ToString();
   }
}
public partial class Add : Expr
{
   public override string PrettyPrint()
   {
      return left.PrettyPrint() + " + " + right.PrettyPrint();
   }
}
```

The initial pretty printer as an operation extension of the initial data variants.

© 2010 Ralf Lämmel

```
public abstract partial class Expr
{
    public abstract int Evaluate();
}
public partial class Const : Expr
{
    public override int Evaluate()
    {
        return info;
    }
}
public partial class Add : Expr
{
    public override int Evaluate()
    {
        return left.Evaluate() + right.Evaluate();
    }
}
```



© 2010 Ralf Lämmel

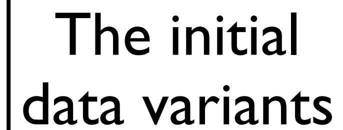
```
public partial class Neg : Expr
{
    public Expr operand;
    public override string PrettyPrint()
    {
        return "- (" + operand.PrettyPrint() +")";
    }
    public override int Evaluate()
    {
        return - operand.Evaluate();
    }
}
```

© 2010 Ralf Lämmel

# Non-solution in C#: Cast-based type switch

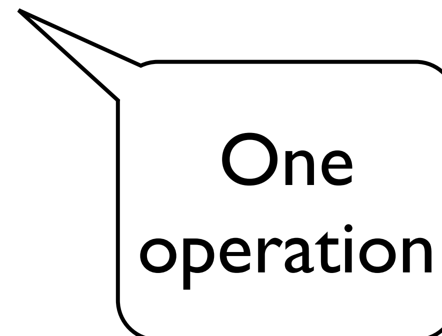
© 2010 Ralf Lämmel

```
public abstract class Expr
{
}
public class Const : Expr
{
    public int info;
}
public class Add : Expr
{
    public Expr left, right;
}
```



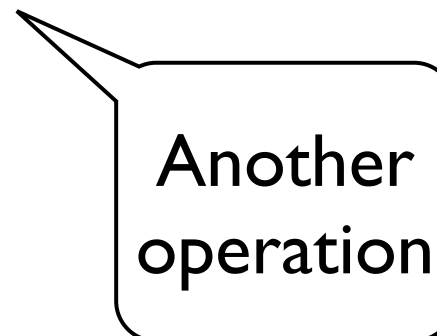
© 2010 Ralf Lämmel

```
public static class Evaluator
{
    public static int Evaluate(Expr that)
    {
        var c = that as Const;
        if (c != null) return c.info;
        var a = that as Add;
        if (a != null) return Evaluate(a.left) + Evaluate(a.right);
        throw new ArgumentException();
    }
}
```



© 2010 Ralf Lämmel

```
public class PrettyPrinter
{
   public virtual string PrettyPrint(Expr that)
   {
      var c = that as Const;
      if (c != null) return c.info.ToString();
      var a = that as Add;
      if (a != null) return PrettyPrint(a.left) + "+" + PrettyPrint(a.right);
      throw new ArgumentException();
   }
}
```



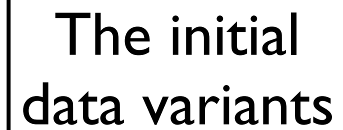
```
public class Neg : Expr
{
   public Expr operand;
}
public class PrettyPrinterWithNeg : PrettyPrinter
{
   public override string PrettyPrint(Expr that)
   {
      try { return base.PrettyPrint(that); }
      catch (ArgumentException)
      {
         var n = that as Neg;
         if (n != null) return "- (" + PrettyPrint(n.operand) + ")";
         throw new ArgumentException();
      }
   }
}
```

© 2010 Ralf Lämmel

# Non-solution in C#: Extension methods

© 2010 Ralf Lämmel

```
public abstract class Expr
{
}
public class Const : Expr
{
   public int info;
}
public class Add : Expr
{
   public Expr left, right;
}
```



© 2010 Ralf Lämmel

One operation

```
public static class PrettyPrinter
{
    public static string PrettyPrint(this Const that)
    {
        return that.info.ToString();
    }
    public static string PrettyPrint(this Add that)
    {
        return that.left.PrettyPrint() + " + " + that.right.PrettyPrint();
    }
}
```

Requires dispatch!

© 2010 Ralf Lämmel

# 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 to achieve both dimensions of extensibility (and separate compilation and static type safety)?

© 2010 Ralf Lämmel

# Further reading

- Phil Wadler's seminal email on the problem

  http://www.daimi.au.dk/~madst/tool/papers/expression.txt
- Clever encodings (Torgersen, ECOOP 2004)
- Open classes (AspectJ et al.)
- Expanders (Warth et al., OOPSLA 2006)
- JavaGI (Wehr et al., ECOOP 2007)
- **Haskell's type classes** (Lämmel, Ostermann, GPCE 2006)
- ...

© 2010 Ralf Lämmel

# Thanks!
# Questions and comments welcome.