Independently Extensible Solutions to the Expression Problem

Martin Odersky, EPFL

Matthias Zenger, Google

History

The expression problem is fundamental for software extensibility.

- It arises when recursively defined datatypes and operations on these types have to be extended simultaneously.
- It was first stated by Cook [91], named as such by Wadler [98].
- Many people have worked on the problem since.

Problem Statement

Suppose we have

- a recursively defined datatype, defined by a set of cases, and
- processors which operate on this datatype.

There are two directions which we can extend this system:

1. Extend the datatype with new data variants,
2. Add new processors.

Problem Statement (2)

Find an implementation technique which satisfies the following:

- Extensibility in both dimensions: It should be possible to add new data variants *and* processors.
- Strong static type safety: It should be impossible to apply a processor to a data variant which it cannot handle.
- No modification or duplication: Existing code should neither be modified nor duplicated.
- Separate compilation: Compiling datatype extensions or adding new processors should not encompass re-type-checking the original datatype or existing processors.

New concern in this paper:

- Independent extensibility: It should be possible to combine independently developed extensions so that they can be used jointly.

Scenario

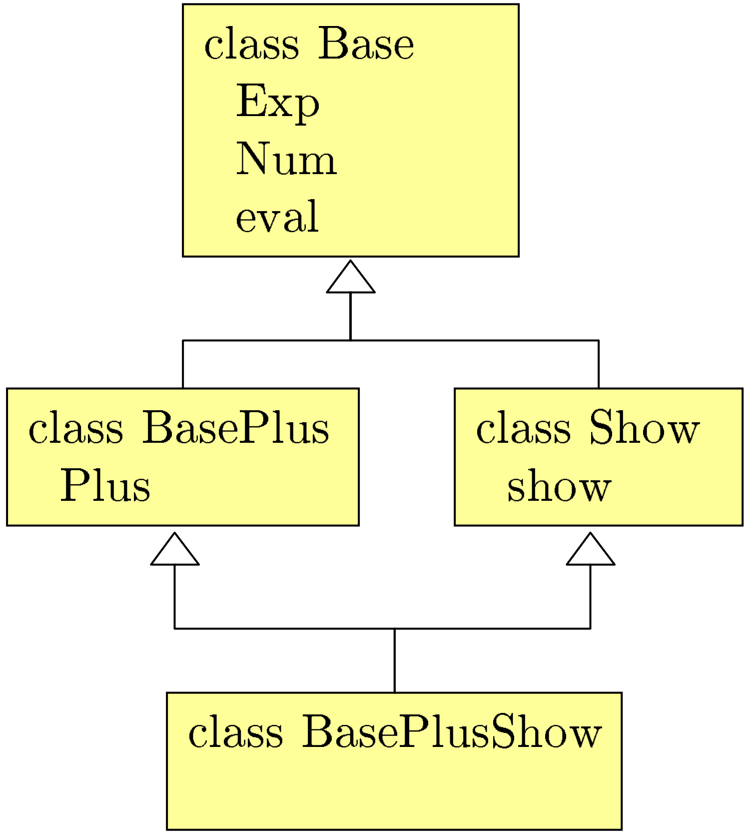
Say, we have

- a base type Exp for expressions, with an operation eval.
- a concrete subtype Num of Exp, representing integer numbers.

We now want to extend this system with

- a new expression type: Plus
- a new operation: show

Finally, we want to combine both extensions in one system.



State of the Art 10 Years Ago

Two canonical structuring schemes each support extension in one dimension, but prevent extension in the other.

1. A data centric structure (using virtual methods) enables addition of new kinds of data.

2. An operation centric structure (using pattern matching or visitors) enables addition of new kinds of operations.

More refined solutions often build on one of these schemes.

State of the Art Today

Many people have proposed partial solutions to the expression problem:

- By allowing a certain amount of dynamic typing or reflection: extensible visitors (Krishnamurti, Felleisen, Friedman 98), walkabouts (Palsberg and Jay 97).
- By allowing default behavior: multi-methods (MultiJava 2000), visitors with defaults (Odersky, Zenger 2001).
- By deferring type checking to link time (relaxed MultiJava 2003).
- Using *polymorphic variants* (Garrigue 2000)
- Using *ThisType* and *matching* (Bruce 2003)
- Using *generics* with some clever tricks (Torgersen 2004)

In this Paper

- We present new solutions to the expression problem.
- They satisfy all the criteria mentioned above, including independent extensibility.
- We study two new variations of the problem:
  tree transformers and binary methods.
- Two families of solutions: data-centric and operation-centric. Each one is the dual of the other.

- Our solutions are written in Scala.
- They make essential use of the following language constructs:
  - abstract types,
  - mixin composition, and
  - explicit self types (for the visitor solution).

  (These are also the core constructs of the *νObj* calculus).
- Compared to previous solutions, ours tend to be quite concise.
- These solutions were also reproduced in OCaml (Rémy 2004).

To Default or Not Default?

- Solutions to the expression problem fall into two broad categories – with defaults and without.
- Solutions with defaults permit processors that handle unknown data uniformly, using a default case.
- Such solutions tend to require less planning.
- However, often no useful behavior for a default case exists, there's nothing a processor known to do except throw an exception.
- This is re-introduces run-time errors through the backdoor.

A Solution with Defaults

Outer trait defines system in question
Everything else is nested in it.

Base language:

```
trait Base {
  class Exp;
  case class Num(v: int)
    extends Exp
   def eval(e: Exp) = e match {
    case Num(v) => v
  }
}
```

Data extension:

```
trait BasePlus extends Base {
  case class Plus(l: Exp, r: Exp)
    extends Exp;
  def eval(e: Exp) = e match {
    case Plus(l, r)   => eval(l) + eval(r)
    case _            => super.eval(e)
  }
}
```

what if we had forgotten
to override show?

Operation extension:

```
trait Show extends Base {
   def show(e: Exp) = e match {
      case Num(v) => v.toString()
   }
}
```

Combination:

```
trait ShowPlus extends Show with BasePlus {
 override def show(e: Exp) = e match {
  case Plus(l, r)   => show(l) + "+" + show(r)
  case _            => super.show(e)
}}
```

Solutions without Defaults

Solutions without defaults fall into two categories.

- Data-centric: operations are distributed as methods in the individual data types.
- Operation-centric: operations are grouped separately in a visitor object.

Let's try data-centric first.

solution

Problem:
type Exp needs to vary co-variantly
with operation extensions.

Base

```
trait Base {
    trait Exp { def eval: int }
    class Num(v: int) extends Exp {
        val value = v;
        def eval = value
    }}
```

```
trait Ba  s extends Base {
    class Plus(l: Exp, r: Exp)
        extends Exp {
        val left: Exp = l;
        val right: Exp = r;
        def eval = left.eval + right.eval
    }}
```

Operation extension:

```
trait Show extends Base {
    trait Exp extends super.Exp {
        def show: String;
    }
    class Num(v: int) extends
            super.Num(v) with Exp {
        def show = value.toString();
     }}
```

ERROR:
show is not a member of
Base.Exp!

Comb

```
trait ShowPlus e    s BasePlus with Show {
    class Plus(l:    , r: Exp) extends
                su  er.Plus(l, r) with Exp {
        def show =
            left.show + "+" + right.show;
    }}
```

Achieving Covariance

- Covariant adaptation can be achieved by defining an *abstract type.*
- Example:

```
type exp <: Exp;
```

  This defines exp to be an abstract type with upper bound Exp.

- The exp type can be refined co-variantly in subtypes.

Data-centric Solution

Base language:

```
trait Base {
   type exp <: Exp;
   trait Exp { def eval: int }
   class Num(v: int) extends Exp {
      val value = v;
      def eval = value
   }}
```

Data extension:

```
trait BasePlus extends Base {
   class Plus(l: exp, r: exp)
      extends Exp {
      val left: exp = l;
      val right: exp = r;
      def eval = left.eval + right.eval
   }}
```

Operation extension:

```
trait Show extends Base {
   type exp <: Exp;
   trait Exp extends super.Exp {
      def show: String;
   }
   class Num(v: int) extends
         super.Num(v) with Exp {
      def show = value.toString();
    }}
```

Combined extension:

```
trait ShowPlus extends BasePlus with Show {
   class Plus(l: exp, r: exp) extends
               super.Plus(l, r) with Exp {
      def show =
         left.show + "+" + right.show;
   }}
```

Tying the Knot

- Classes that contain abstract types are themselves abstract.
- Before instantiating such a class, the abstract type has to be defined concretely.
- This is done using a type alias, e.g. type exp = Exp;
- For instance, here is a test program that uses the ShowPlus system.

```
object ShowPlusTest extends ShowPlusNeg with Application {
    type exp = Exp;
    val e: Exp = new Plus(new Num(1), new Num(2));
    Console.println(e.show + " = " + e.eval)
}
```

Independent Data Extensions

- Let's add to the system with eval and show another data variant for negated terms.

```
trait ShowNeg extends Show {
    class Neg(t: exp) extends Exp {
        val term = t;
        def eval = - term.eval;
        def show = "-(" + term.show + ")"
    }}
```

- The two extensions ShowPlus and ShowNeg can be combined using a simple mixin composition:

```
trait ShowPlusNeg extends ShowPlus with ShowNeg;
```

Tree Transformer Extensions

- So far, all our operators returned simple data types.
- We now study tree transformers, i.e. operators that return themselves the data structure in question.
- This is in principle as before, except that we need to add factory methods.
- As an example, consider adding an operation dble that, given an expression tree of value v, returns another tree that evaluates to 2*v.

The "Dble" Transformer

```
trait DblePlus extends BasePlus {
    type exp <: Exp;
    trait Exp extends super.Exp {
        def dble: exp;
    }
    def Num(v: int): exp;
    def Plus(l: exp, r: exp): exp;

    class Num(v: int) extends super.Num(v) with Exp {
        def dble = Num(v * 2);
     }
     class Plus(l: exp, r: exp) extends super.Plus(l, r) with Exp {
        def dble = Plus(left.dble, right.dble);
     }
  }
```

Factory methods

Combining "Show" and "Dble"

- Combining two operations is more complicated than a simple mixin composition.
- We now have to combine as well all nested types in a "deep composition".

```
trait ShowDblePlus extends ShowPlus with DblePlus {
    type exp <: Exp;

    trait Exp extends super[ShowPlus].Exp
                    with super[DblePlus].Exp;

    class Num(v: int) extends super[ShowPlus].Num(v)
                            with super[DblePlus].Num(v)
                            with Exp;

    class Plus(l: exp, r: exp) extends super[ShowPlus].Plus(l, r)
                                    with super[DblePlus].Plus(l, r)
                                    with Exp;
}
```

Instantiating Transformers

- Instantiating a system with transformers works as before, except that we now also need to define factory methods.

```
trait ShowDblePlusTest extends ShowDblePlus with Application {
    type exp = Exp;

    def Num(v: int) = new Num(v);
    def Plus(l: exp, r: exp): exp = new Plus(l, r)

    val e: exp = new Plus(new Num(1), new Num(2));
    Console.println(e.dble.eval);
}
```

Summary: Data-centric solutions

- We have seen that we can flexibly extend in two dimensions using a data-centric approach.
- Extension with new operations is made possible by abstracting over the data type `exp`.
- Individual extensions can be merged later using mixin composition.
- Merging two data extensions is easy, requires only a flat mixin composition.
- Merging two operation extensions is harder, since it requires to merge nested classes as well, using a deep mixin composition.

Operation-centric Solutions

- Operation-centric solutions are the duals of data-centric solutions.
- Here, all operations together are grouped in a *visitor object*.

Operation-centric Solution

Base language:

```
trait Base {
    trait Exp { def accept(v: visitor): unit }

    class Num(value: int) extends Exp {
        def accept(v: visitor): unit = v.visitNum(value);
    }

    type visitor <: Vis
    trait Visitor {
        def visitNum(value: int): unit;
    }
    class Eval: visitor extends Visitor {
        var result: int = _;
        def apply(t: Exp): int = { t.accept(this); result }
        def visitNum(value: int): unit = { result = value }
    }
}
```

Solution:
explicit self type

Problem:
Eval.this must conform to visitor

Selftype Annotations

- Scala is one of very few languages where the type of this can be fixed by the programmer using a selftype annotation (OCaml is another).
- Type-soundness is maintained by two requirements
  - Selftypes vary covariantly in the class hierarchy.

    I.e. the selftype of a class must be a subtype of the selftypes of all its superclasses.
  - Classes that are instantiated to objects must conform to their selftypes.
- Selftype annotations are not the same thing as Bruce's *mytype*, since they do not vary automatically.

Operation-centric Solution (2)

Base language:

```
trait Base {
  trait Exp { def accept(v: visitor): unit }

  class Num(value: int) extends Exp {
    def accept(v: visitor): unit = v.visitNum(value);
  }

  type visitor <: Visitor;
  trait Visitor {
    def visitNum(value: int): unit;
  }
  class Eval: visitor extends Visitor {
    var result: int = _;
    def apply(t: Exp): int = { t.accept(this); result }
    def visitNum(value: int): unit = { result = value }
  }
}
```

Data extension:

```
trait BasePlus extends Base {
  type visitor <: Visitor;
  trait Visitor extends super.Visitor {
    def visitPlus(left: Exp, right: Exp): unit;
  }
  class Plus(left: Exp, right: Exp) extends Exp {
    def accept(v: visitor): unit =
      v.visitPlus(left, right);
  }
  class Eval: visitor extends
      super.Eval with Visitor {
    def visitPlus(l: Exp, r: Exp): unit =
      result = apply(l) + apply(r);
  }
}
```

Operation-centric Solution (3)

Base language:

```
trait Base {
   trait Exp { def accept(v: visitor): unit }

   class Num(value: int) extends Exp {
      def accept(v: visitor): unit = v.visitNum(value);
   }

   type visitor <: Visitor;
   trait Visitor {
      def visitNum(value: int): unit;
    }
   class Eval: visitor extends Visitor {
      var result: int = _;
      def apply(t: Exp): int = { t.accept(this); result }
      def visitNum(value: int): unit = { result = value }
    }
}
```

Data extension:

```
trait BasePlus extends Base {
  type visitor <: Visitor;
  trait Visitor extends super.Visitor {
    def visitPlus(left: Exp, right: Exp): unit;
  }
  class Plus(left: Exp, right: Exp) extends Exp {
    def accept(v: visitor): unit =
      v.visitPlus(left, right);
  }
  class Eval: visitor extends
      super.Eval with Visitor {
    def visitPlus(l: Exp, r: Exp): unit =
      result = apply(l) + apply(r);
  }
}
```

Operation extension:

```
trait Show extends Base {
   class Show: visitor extends Visitor {
      var result: String = _;
      def apply(t: Exp): String = { t.accept(this); result }
      def visitNum(value: int): unit =
         { result = value.toString() }
}
```

Operation-centric Solution (4)

Base language:

```
trait Base {
  trait Exp { def accept(v: visitor): unit }

  class Num(value: int) extends Exp {
    def accept(v: visitor): unit = v.visitNum(value);
  }

  type visitor <: Visitor;
  trait Visitor {
    def visitNum(value: int): unit;
  }
  class Eval: visitor extends Visitor {
    var result: int = _;
    def apply(t: Exp): int = { t.accept(this); result }
    def visitNum(value: int): unit = { result = value }
  }
}
```

Data extension:

```
trait BasePlus extends Base {
  type visitor <: Visitor;
  trait Visitor extends super.Visitor {
    def visitPlus(left: Exp, right: Exp): unit;
  }
  class Plus(left: Exp, right: Exp) extends Exp {
    def accept(v: visitor): unit =
      v.visitPlus(left, right);
  }
  class Eval: visitor extends
      super.Eval with Visitor {
    def visitPlus(l: Exp, r: Exp): unit =
      result = apply(l) + apply(r);
  }
}
```

Operation extension:

```
trait Show extends Base {
  class Show: visitor extends Visitor {
    var result: String = _;
    def apply(t: Exp): String = { t.accept(this); result }
    def visitNum(value: int): unit =
      { result = value.toString() }
}
```

Combined extension:

```
trait ShowPlus extends Show with BasePlus {
  class Show: visitor extends super.Show {
    def visitPlus(l: Exp, r: Exp): unit =
      result = apply(l) + "+" + apply(r);
  }
}
```

Summary: Operation-centric solutions

- Operation-centric is the dual of data-centric. Both approaches can extend in two dimensions.
- Extension with new *data* is made possible by abstracting over the data type visitor.
- Individual extensions are again merged using mixin composition.
- Explicit selftypes are needed to pass a visitor along the tree.
- Now, merging two *operation extensions* is easy, requires only a flat mixin composition.
- Merging two *data extensions* is harder, since it requires to merge nested classes as well, using a deep mixin composition.
- So in a sense, we have made the two approaches more compatible, but we have not eliminated their differences.

Conclusion

- We have developed two dual families of solutions to the expression problem in Scala.
- New variants: Tree transformers, binary methods (see paper).
- New concern: Independent extensibility.
- Solutions use standard technology (in the Scala world), which shows up in almost every component architecture.
  - abstract types
  - mixin composition
  - explicit selftypes
- This further strengthens the conjecture that the expression problem is indeed a good representative for component architecture in general.