The Expression Problem
Scala vs other languages
Part I

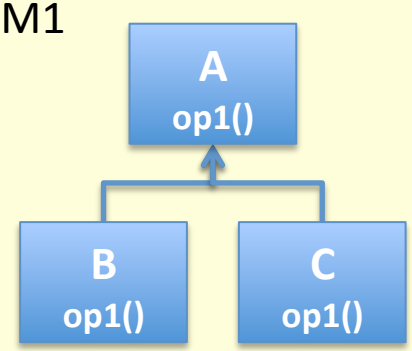
Görel Hedin
Dept of Computer Science, Lund University

These slides:
http://fileadmin.cs.lth.se/sde/people/gorel/misc/ExpProblemTalkPart1.pdf

Scala course, Lund, June 14, 2012

What is the expression problem?

M1



M3

Add new operation op2 to A, B and C

M2

Add a new subtype D with an op1 impl.

M4

Combine extensions. Add op2 for D.

Extend a type with both subtypes and operations

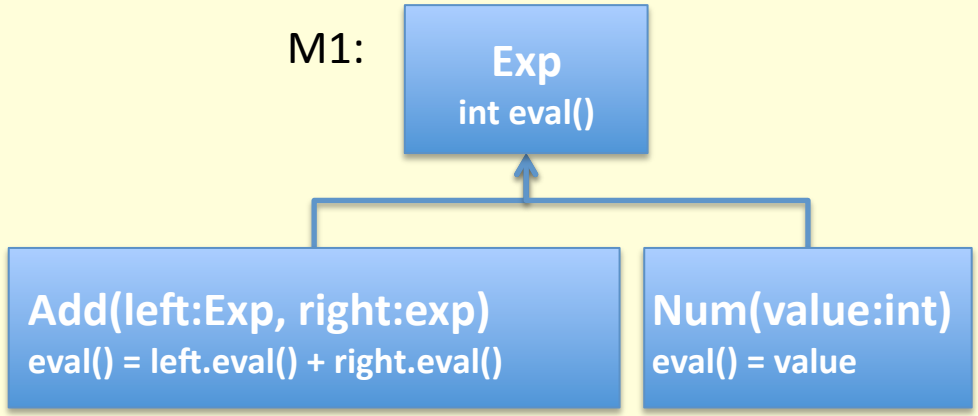
EP Criteria

Criterion	… that is …
Source-level modularity	No modification of existing modules
Binary-level modularity	Separate compilation
Static typechecking	No unguarded casts needed
Simple design	No duplication, no boilerplate, …
Free module combination	No linearization needed

[Reynolds75], [GoF94], [Wadler98], [Zenger&Odersky04], [Ernst04], [Clifton06], …

The standard problem
[Wadler98, Ernst04]

M1:



M3
Add new operation: show()
E.g., for Add:
show() =
left.show() + ”+” + right.show()

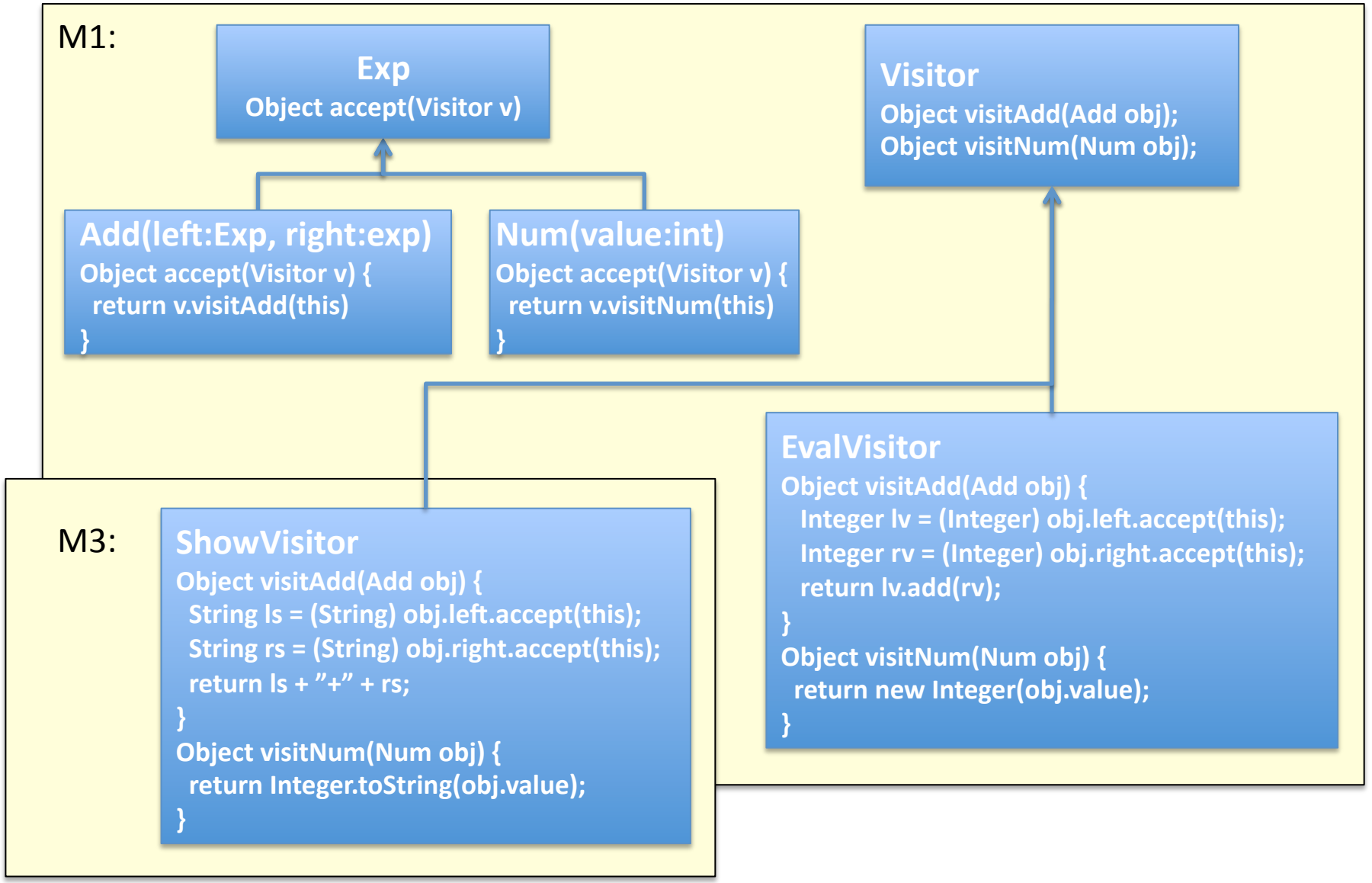
M2:
Add a new subtype Neg
with an eval() impl.

M4:
Combine extensions.
Add show() for Neg.

Typical solutions

- Visitors: Basic OOP (Java, C++, …)
- Open classes: AspectJ, MultiJava, JastAdd
- Hierarchical classes: gbeta, Scala, …

The Visitor Solution [GoF94]



The EP criteria for Visitor

Criterion	Visitor solution
Source-level modularity	Yes
Binary-level modularity	Yes
Static typechecking	Ungarded casts used
Simple design	Lots of boilerplate code
Free module combination	Problematic

The Open Class solution (JastAdd syntax)

M1.ast

```
abstract Exp;
Num : Exp ::= <value:int>;
Add : Exp ::= Left:Exp Right:Exp;
```

M1.jrag

```
aspect M1 {
  public abstract int Exp.eval();
  public int Num.eval() {return getvalue();}
  public int Add.eval() {
    return getLeft().eval() + getRight().eval();
  }
}
```

M2.ast

```
Neg : Exp ::= Exp;
```

M2.jrag

```
aspect M2 {
  public int Neg.eval() {
    return -getExp().eval();
  }
}
```

M3.jrag

```
aspect M3 {
  public abstract String Exp.show ();
  public String Num.show() {
    return Integer.toString(getvalue());
  }
  public String Add.show() {
    return getLeft().show() + "+" + getRight().show();
  }
}
```

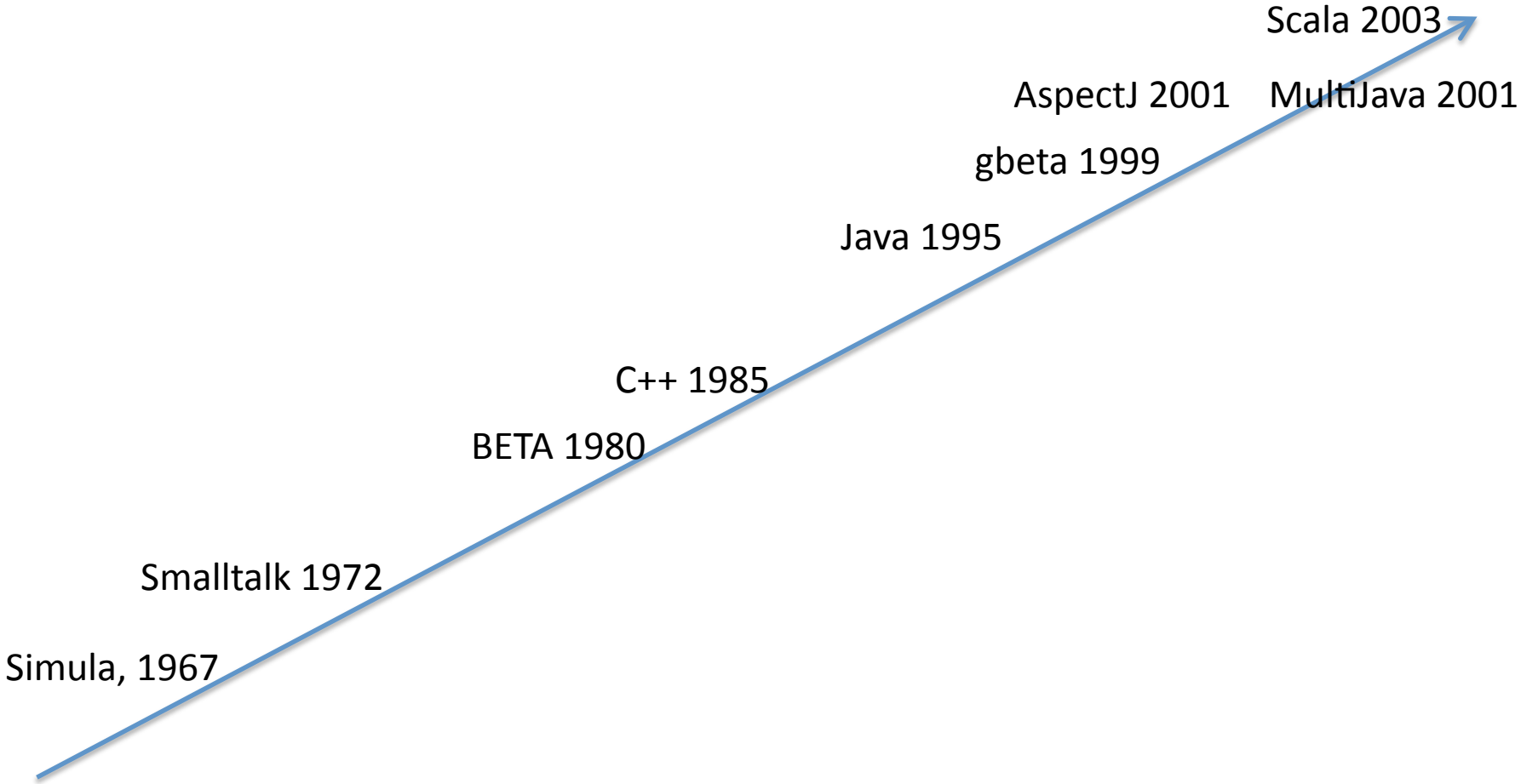
M4.jrag

```
aspect M4 {
  public String Neg.show() {
    return "-(" + getExp().show() + ")";
  }
}
```

The EP criteria for Open classes

Criterion	JastAdd	MultiJava, AspectJ
Source-level modularity	Yes	Yes
Binary-level modularity	No. Generates Java from full specification.	Yes
Static typechecking	Yes, but only on the generated Java code.	Yes
Simple design	Yes	Yes
Free module combination	Yes	Yes

Some OO programming languages

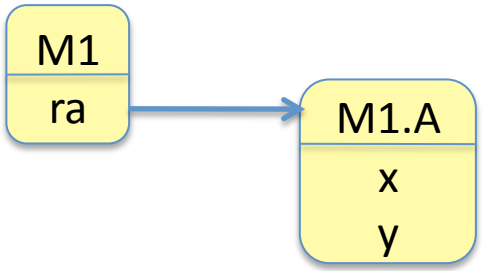


Virtual classes in BETA

(using Java-like syntax)

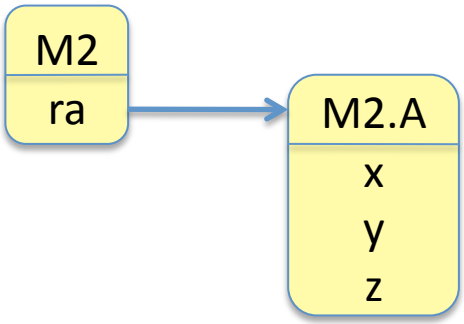
```
class M1 {
  virtual class A {
    int x, y;
  }
  A ra = new A;
}
```

- Class M1 contains a class A.
- Class A is *virtual*.
- This means that A objects have *at least* fields x and y.




```
class M2 extends M1 {
  refine class A {
    int z;
  }
}
```

- Class M2 contains a *refinement* of A.
- The refinement extends A with a z field.
- Note that what object is created at ”new A” depends on if it is executed in an M1 or an M2 object.



- A is a *virtual type*. This means that the actual type of A depends on the containing object.

The gbeta solution (using Java-like syntax)

```
class M1 {
  virtual class Exp {
    int eval() {}
  }
  virtual class Num extends Exp {
    int value;
    int eval { return value; }
  }
  virtual class Add extends Exp {
    Exp left,right;
    int eval {
      return left.eval()+right.eval();
    }
}
```

```
class M3 extends M1{
  refine class Exp {
    String show() {}
  }
  refine class Num{
    String show() {
      return Integer.toString(value);
    }
  }
  refine class Add {
    String show() {
      return left.show()+"+"+right.show();
    }
  }
}
```

```
class M2 extends M1 {
  virtual class Neg extends Exp {
    Exp exp;
    int eval() { return –exp.eval() }
  }
}
```

```
class M4 extends M2, M3 {
  refine class Neg {
    String show() {
      return "-(" + exp.show() + ")";
    }
  }
}
```

The EP criteria for gBeta

Criterion	gBeta
Source-level modularity	Yes
Binary-level modularity	Yes
Static typechecking	Yes
Simple design	Yes
Free module combination	Yes

A simple Scala solution

```
trait M1 {
  trait Exp { def eval: Int }
  class Num(value: Int) extends Exp {
    def eval = value
  }
  class Add(left: Exp, right: Exp) extends Exp {
    def eval = left.eval + right.eval
  }
}
```

```
trait M3 extends M1 {
  trait Exp extends super.Exp {
    def show: String
  }
  class Num(value: Int)
        extends super.Num(value) with Exp {
    def show= value.toString()
  }
  class Add(left: Exp, right: Exp)
        extends super.Add(left, right) with Exp {
    def show= left.show + "+" + right.show
  }
}
```

```
trait M2 extends M1 {
  class Neg(term: Exp) extends Exp {
    def eval = -term.eval
  }
}
```

```
trait M4 extends M2 with M3{
  class Neg(term: Exp)
        extends super.Neg(term) with Exp {
    def show = "-(" + term.show + ")"
  }
}
```

boilerplate code

Problems?

- The previous Scala solution works for my simple examples.
- But the solution given by Zenger and Odersky is more complex:
  - An abstract type *exp* is introduced
  - All fields are typed by *exp*, rather than by the class *Exp*
- Why? (Exercise)

Zenger&Odersky's Scala solution

```
trait M1 {
  type exp <: Exp;
  trait Exp { def eval: Int }
  class Num(value: Int) extends Exp {
    def eval = value
  }
  class Add(left: exp, right: exp) extends Exp {
    def eval = left.eval + right.eval
  }
}
```

```
trait M2 extends M1 {
  class Neg(term: exp) extends Exp {
    def eval = -term.eval
  }
}
```

```
trait M3 extends M1 {
  type exp <: Exp;
  trait Exp extends super.Exp {
    def show: String
  }
  class Num(value: Int)
        extends super.Num(value) with Exp {
    def show= value.toString()
  }
  class Add(left: exp, right: exp)
        extends super.Add(left, right) with Exp {
    def show= left.show + "+" + right.show
  }
}
```

```
trait M4 extends M2 with M3{
  class Neg(term: exp)
        extends super.Neg(term) with Exp {
    def show = "-(" + term.show + ")"
  }
}
```

```
object M4Test extends M4 with App{
  type exp = Exp;
  val e = new Add(new Neg(new Lit(2)), new Lit(11))
  println(e.show)
}
```

The EP criteria for Scala

Criterion	Scala
Source-level modularity	Yes
Binary-level modularity	Yes
Static typechecking	Yes
Simple design	Extra abstract type. Some small amounts of boilerplate
Free module combination	Yes

Exercises

- Write a module M5 that adds a subtype Mul (multiplication) with the *eval* operation.
- Write a module M6 that adds the operation *show* to the Mul type.
- Write a module M7 that combines Add, Num and Mul (but without Neg)
- Write a module M8 that adds a new operation *print*, that prints the expression tree in some way on standard output, using indentation to show the tree structure.

- If possible, explain why the abstract type is needed. Construct an example that does not work without it. Email it to gorel@cs.lth.se

- M1..M4 modules, with test programs, are available for the Simple and the Zenger-Odersky solutions here:
  - http://fileadmin.cs.lth.se/sde/people/gorel/misc/ScalaCodeSimple.zip
  - http://fileadmin.cs.lth.se/sde/people/gorel/misc/ScalaCodeZengerOdersky.zip

- Zenger and Odersky's original code is available here, for reference: http://lampwww.epfl.ch/~odersky/papers/ExpressionProblem