Challenging Typing Issues in Object-Oriented languages

Kim B. Bruce
Williams College
Difficult Typing Issues

• Interaction of mutually dependent classes
• Look at concrete problems
• “Patterns” provide good examples
  - Identified as important
  - Little discussion of inheritance
• Some solutions - some open
• No typing rules - lots of code
Expression Problem

- Operations on expressions of language
- Values represent terms of language
- Operators represent language processors:
  - Interpreter
  - Pretty-printer
- Concerns about extensibility
History

- Wadler - Java Genericity list (1998)
  - Cartwright, Bruce
- Krishnamurthi, Felleisen & Friedman (1998)
- Cook (1990)
- Reynolds (1975)
ML Solution

datatype term = Const of int | Neg of term | Plus of term*term

fun interp (Const n) = n
  | interp (Neg t) = - (interp t)
  | interp (Plus t u) = (interp t) + (interp u)

Easy to add new functions:

fun prettyPr (Const n) = ...
  | prettyPr (Neg t) = “-” ^ (prettyPr t)
  | prettyPr (Plus t u) =
                            (prettyPr t) ^ “+” ^ (prettyPr u)

Hard to add new terms!
interface Form {
    int interp(); // Interpret formula
}

class ConstForm implements Form {
    int value; // value of constant

    ConstForm(int value) {
        this.value = value;
    }

    int interp() { return value; }
}

class NegForm implements Form {...}
Adding New Formula Easy

class PlusForm implements Form {
    Form first, second;

    PlusForm(Form firstp, Form secondp) {
        first = firstp;
        second = secondp;
    }

    int interp() {
        return first.interp() + second.interp();
    }
}
Adding new method harder

interface PPForm extends Form {
    String prettyPrint();
}

class PPConstForm extends ConstForm implements PPForm {

    PPConstForm(int value) {
        super(value);
    }

    String prettyPrint() {
        return "" + value;
    }
}

WOODS '03
Type problems can arise!

class PPPlusForm extends PlusForm
    implements PPForm {
    // Form first, second;
    PPPlusForm(PPForm firstp, PPForm secondp) {
        super(firstp, secondp);
    }

    String prettyPrint() {
        return "(" + first.prettyPrint() + " + " + " + " + second.prettyPrint() + ")";
    }
}

Type error - Form not support prettyPrint!
Need type casts

class PPPlusForm extends PlusForm
    implements PPForm {

    PPPlusForm(PPForm firstp,
               PPForm secondp) {
        super(firstp,secondp);
    }

    String prettyPrint() {
        return ((PPForm)first).prettyPrint() + " + " +
               ((PPForm)second).prettyPrint();
    }
}
class PlusForm implements @Form {
    ThisType first, second;

    PlusForm(ThisType firstp,
             ThisType secondp) {
        first = firstp;
        second = secondp;
    }

    int interp() {
        return first.interp() + second.interp();
    }
}

ThisType represents interface of class - Form
Solves Typing Problem

class PPPlusForm extends PlusForm implements @PPForm {

    PPPlusForm(ThisType firstp, ThisType secondp) {
        super(firstp, secondp);
    }

    String prettyPrint() {
        return first.prettyPrint() + " + " + second.prettyPrint();
    }
}

ThisType represents PPForm here!
Still Not Ideal Solution

- Class / interface depends on operations
  - Must explicitly extend all classes when add operation
  - Expression w/ just interp method (Form) different from expression w/ both (PPForm).
  - Applications must be rewritten to take new classes when add operation

- Advantages / disadvantages opposite ML.
- Regain ML advantages with Visitor pattern
Visitor Pattern

- Supports data structures that may have different “visitors” operating on them
- Easy to add operations
- Hard to add new variants
- In standard Java visitors return Object, so need lots of casts
- Write in GJ / LOOJ using parametric polymorphism to avoid casts.
Expressions w/ Visitors

interface Form {
    // Process formula with visitor lp.
    <Result> Result process(BasicLangProc<Result> lp);
}

// BasicLangProc: "Visitor" of consts & negations
interface BasicLangProc<Result> {
    // process constant expression
    Result constCase(ConstForm cf);

    // process negation expression
    Result negCase(NegForm nf);
}
Classes taking Visitor

class ConstForm implements Form {
    int value;    // value of constant
    ConstForm(int val) { value = val;}

    Result process(BasicLangProc<Result> lp) {
        return lp.constCase(this); }
}

class NegForm<Result> implements Form<Result> {
    Form pos;    // formula to be negated
    ...
    Result process(BasicLangProc<Result> lp) {
        return lp.negCase(this); }
}
Visitors for Form

class BasicInterp implements BasicLangProc<Integer> {
    Integer constCase(ConstForm cf) {
        return new Integer(cf.value); }

    Integer negCase(NegForm nf) {
        return new Integer(
            -(nf.pos.<Integer>process(this)).intValue());}
}

class BasicPPer implements BasicLangProc<String> { 
    ...
    String negCase(NegForm nf) {
        return “-” + nf.pos.<String>process(this); }
}
Using Visitors

Form cf = new ConstForm(17);
Form nf = new NegForm(cf);

BasicLangProc<Integer> binterp =
    new BasicInterp();

... nf.<Integer>process(binterp) ...;

BasicPPer<String> bpper = new BasicPPer();

... nf.<String>process(bппer) ...;
Tracing Computation

nf.<Integer>process(binterp)
calls
binterp.negCase(this) where this is nf
returns
new Integer(
  - nf.pos.<Integer>process(this).intVal()
) where nf.pos is cf and this is binterp and
cf.<Integer>process(binterp)
returns
new Integer(17)
returns
new Integer(-17)
Add Plus Expression

```java
interface LangProc<Result>
    extends BasicLangProc<Result>{
    Result plusCase(PlusForm pf);
}

public class PlusForm implements Form {
    Form first, second;  // sum parts

    PlusForm(Form fstp, Form sndp) {
        first = fstp; second = sndp; }

    <Result>Result process(BasicLangProc<Result> lp) {
        return ((LangProc<Result>)lp.plusCase(this)); }
}
```
Extended Interpreter

```java
public class Interp extends BasicInterp implements LangProc<Integer> {

    // return sum of two pieces.
    public Integer plusCase(PlusForm pf) {
        int firstVal = (pf.first.<Integer>process(this)).intValue();
        int secondVal = (pf.second.<Integer>process(this)).intValue();

        return new Integer(firstVal + secondVal);
    }
}
```
Evaluating Visitors

- Easy to add visitors (new methods)
  - Polymorphism helps avoid most casts

- Harder to add new variants
  - Must extend all existing visitors with a new case
  - But easier than ML because of inheritance
Making Visitors Type-Safe?

- Can we eliminate type cast?
  - Want statically typed solution!

- Solution: change type bound in methods by adding new type parameters to Form.

- Add
  - Visitor type parameter to Form represents language processor type for formula
Making Visitors Type-Safe

```java
interface Form<Visitor extends BasicLangProc> {
    // Process formula with visitor lp.
    <Result> Result process(@Visitor<Result> lp);
}

interface BasicLangProc<Result> {
    // process constant expression
    Result constCase(@ConstForm<ThisTypeFcn> cf);

    // process negation expression
    Result negCase(@NegForm<ThisTypeFcn> nf);
}
```
Making Visitors Type-Safe

class NegForm<Visitor extends BasicLangProc>
    implements Form<Proc> {
    @Form<Proc> pos; ...
    <Result> Result process(@Visitor<Result> lp) {
        return lp.negCase(this); }
}

class PlusForm<Proc extends LangProc>
    implements Form<Proc> {
    @Form<Proc> left, right; ...
    <Result> Result process(@Visitor<Result> lp) {
        return lp.plusCase(this); }
}
Was It Worth It?

• Make type-safe, *but only with great pain!*
  
  - Need bounded polymorphism on type-fcns
  
  - `ThisTypeFcn`
  
  - *This* constructor w/ more complex expressions (or Factory object specified w/ `ThisType`)
  
  - Alternative using contravariant type functions

• Slightly(!) easier typing if make Form (rather than process) parameterized on Result

  - Use `ThisType` rather than `ThisTypeFcn`
New Approach

- Mutually recursive.
- Form type depend on LangProc type and vice-versa.
- Follow approach of Bruce-Vanderwaart (MFPS ‘99) as implemented in LOOM
- Useful w/ other patterns: Subject-Observer
Type Groups

group Basic {
    interface Form {
        <Result> Result process(@LangProc<Result> lp);}

    interface LangProc<Result> {
        Result constCase(@ConstForm cf);
        Result negCase(@NegForm nf); }

    ...

class NegForm implements @Form {
    @Form pos;   // value of constant
    ...
    <Result> Result process(@LangProc<Result> lp) {
        return lp.negCase(this); }
}

WOODS '03
Basic group (cont)

class Interp implements @LangProc<Integer> { 
    ... 
    Integer negCase(@NegForm nf) { return new Integer(nf.pos.<Integer>process(this)); 
    }
}

class PPer implements @LangProc<String> { 
    String negCase(@NegForm nf) { 
        return "-" + nf.pos.<String>process(this);
    }
}

Extending Group

group Full extends Basic {
    interface LangProc<Result> { // extends old
        Result plusCase(@PlusForm pf); }

    class PlusForm implements @Form { ...
        <Result> Result process(@LangProc<Result> lp) {
            return lp.plusCase(this); } }

    class Interp implements @LangProc<Integer>{
        Integer plusCase(@PlusForm pf) {
            return new Integer(
                pf.first.<Integer>process(this)+...); }
    }
}
Using Groups

@Basic.Form bconstf = new Basic.ConstForm(12);
@Basic.PPer bpp = new Basic.PPer();
... bconstf.<String>process(bpp)) ...;

@Basic.Interp binterp = new Basic.Interp();
... bconstf.<Integer>process(binterp) ...;

@Full.Form fcon1 = new Full.ConstForm(17);
@Full.Form fcon2 = new Full.ConstForm(13);
@Full.Form fplus = new Full.PlusForm(fcon1, fcon2);

@Full.Interp finterp = new Full.Interp();
... plusf.<Integer>process(finterp) ...;
Advantages of groups

• Mutually recursive (Wadler’s problem)
• Extending type group extends all interfaces and classes simultaneously
• Easy to add new interfaces & classes
• Need to ensure don’t mix and match
  - Exact types!
• Formulas not depend on result type
• Many other examples where useful
Semantically, ...

- **OO semantics given by fixed points:**
  - Classes represent extensible generators
  - Object types are fixed points of type generators

- **Groups represent mutual recursion**
  - Set of classes in group represents extensible collection of generators
  - Object types are parts of mutually recursive fixed points of type generators.
Can We Preserve More?

group GUIComponents {
    interface Component { ... }
    interface Button extends Component {...}
    interface Window extends Component {
        void addComponent(Component item) {...}
    }
}

group ColorComponents extends GUIComponents {
    interface Component {
        void setColor(Color newColor);
    }
}
Can We Preserve Relations?

ColorComponents.Window clrWindow;
ColorComponents.Button clrButton;

clrWindow.addComponent(clrButton);

Type-safe?
Does clrButton have a setColor method?

What about polymorphism?

<CompGP extends GUIComponents> void doGUIStuff
   (@CompGp.Button button, @CompGp.Window wind) {
      ... wind.addComponent(button)...
   }
What Is Done?

- Parametric Polymorphism for classes and methods: GJ / LOOJ
- ThisType and exact types - LOOJ
- Type groups in LOOM (w/o polymorphism)
  - Modular static typing
  - Ideas like those for MyType
  - Need exact types so don’t mix items from different groups!
Still Open

- Adding groups to LOOJ
- Preserving other relations
- Pure statically-typed solution to Visitor:
  - Expression classes not parameterized by Result
  - No casts (but understandable)!
  - Can we do better than higher-order bounded polymorphism with ThisTypeFcn?
Other Related Work

• Virtual Classes - Beta and gBeta
  - Family polymorphism (disownment)
• Odersky et al - Scala
• Krishnamurthi et al on similar problems
• Findley & Flatt - mixins w/ fixed points
Summary

• Interesting typing problems open
• Systems of interacting objects
• Patterns useful source of ideas
• Avoid specialized solutions
  - Must be of sufficient power and generality to be worth adding to a language.
The End!