Mostly Modular Compilation of Crosscutting Concerns by Contextual Predicate Dispatch

Shigeru Chiba
Salikh Zakirov
Atsushi Igarashi

Tokyo Institute of Technology
Kyoto University
Aspect Oriented Programming

• ... modularizes the un-modularizable.
• ... breaks modularity.
Object Oriented Programming

- Modular type checking and Separate compilation
  - Yes if super classes are given

- Modular reasoning
  - Fairly? but dynamic dispatch
Object-Oriented Programming

Aspect Oriented

• Modular type checking and Separate compilation
  – Yes if super classes are given

• Modular reasoning
  – Fairly? but dynamic dispatch
GluonJ

• A simple AOP extension to Java
  – A subset of AOP functionality of AspectJ

  – An OOP-like mechanism
    • Enhanced method overriding and dispatching

    • For ease of comparison between AOP and OOP
    • Are type checking and compilation modular?
AOP functionality (1)

• Destructive extension
  – An advice can **obliviously** modify a method.
  • Unlike subclass, mixin, traits, ...

```java
class AddExpr {
    Value eval() { ... }
}

Value around(AddExpr ae):
  execution(Value AddExpr.eval())
  && this(ae) {
    if (…) proceed();
    else ... ;
  }
```
AOP functionality (2)

- Limited scope
  - An advice can modify a method call in a body
  - Breaking modularity?

```java
class VarDecl {
    Value init() {
        v = right.eval();
    }
}

aspect Logging {
    before():
        call(void Expr.eval())
        && withincode(* VarDecl.init()) {
            ...
        }
}

class AddExpr {
    Value eval() { ... }
}
```
GluonJ:

- Destructive extension
  - A reviser can add and override a method, and add a field to an existing class.
  - It cannot have an explicit constructor.

```java
class AddExpr {
    Value eval() {
        ... }
}

class FloatEx revises AddExpr {
    Value eval() {
        if (...) super.eval();
        else ...
    }
}
```
GluonJ: A within method

- Limited scope
  - A method may have a predicate.
  - Its method overriding is effective only when it is called from ...

```java
class VarDecl {
    Value init() {
        v = right.eval();
    }
}
class AddExpr {
    Value eval() { ... }
}

class Log revises AddExpr {
    Value eval()
        within VarDecl.init() {
            ...
        }
}  
```
Contextual predicate dispatch

• GluonJ
  – Predicates refers to non-local contexts i.e. within who is a caller.
    • Currently only within is available.
  – to deal with crosscutting concerns

• Original predicate dispatch
  – Predicates refers to only local contexts such as arguments and receiver’s fields
    • for unambiguity and exhaustiveness
GluonJ:

- Some revisers may modify the same.
  - Like mixin, they are linearized and applied one by one.
  - \textbf{requires} must be a total order

```java
class AddExpr {
    Value eval() { ... }
}

class FloatEx revises AddExpr {
    Value eval() { ... }
}

class StringEx requires FloatEx revises AddExpr {
    Value eval() {
        if (...) super.eval();
        else ... ;
    }
}
```
GluonJ:

- How to access revised methods/fields
  - Effects by a required reviser is visible.
  - A using declaration make them visible from classes.

```java
class AddExpr {
    Value eval() { ... }
}

class Printing revises AddExpr {
    void print() { ... }
}
```

```java
using Printing;

class Printer {
    static void print(AddExpr e) {
        e.print();
    }
}
```
Core Calculus of GluonJ

• An extension of Featherweight Java
  – to rigorously discuss modular compilation

• Syntax

\[
CL ::= \text{class } C \text{ extends } C \text{ using } R \{ \ C f \ ; \ M \ \}
          | \text{class } R \text{ revises } C \text{ using } R \{ \ M \ \}
L ::= \ C \ | \ R
M ::= \ C \ m(C ; x) \{ \text{return } e; \} \ [ \text{within } L \ ]
\ e ::= \ x | \ e.f | \ e.m(e) | \text{new } C(e) | \ e \text{ in } L
v,w ::= \text{new } C(v)
\]
Type judgment for methods

- **T-METHODR**

  \[ R \; \text{rev} \; D \quad R; \overline{x}: \overline{C}, \; \text{this}: D \vdash e_0 : E_0 \quad E_0 \ll C_0 \]

  for any \( L \), if \( \text{mtype}(m, \text{next}(R), \text{dom}(RT), L) = \overline{D} \rightarrow D_0 \),

  then \( \overline{C} = \overline{D} \) and \( C_0 = D_0 \)

  \[ C_0 \; m(\overline{C} \; \overline{x}) \{} \; \text{return} \; e_0; \; \} \; \text{[within} \; L' \text{]} \; \text{OK} \; \text{IN} \; R \]

  \[ (\text{T-METHODR}) \]

```java
class BinExpr {}

class StringEx revises BinExpr {
  String eval() { ... }
}

class AddExpr ex. BinExpr {}

class FloatEx revises AddExpr {
  float eval() { ... }
}
```
Discussion

• Mostly modular type checking because of this global analysis:
  – `TMethodR` requires all the revisers are known at compile time.
  – `requires` must be a total order on every target class.
Implementation (1)

• 1st stage: translation
  – source code to Java bytecode
  – a reviser becomes a Java class.
  – Inserting type casts

```java
using Printing;
class Printer {
  static void print(AddExpr e) {
    e.print();
  }
}
((Printing)e).print();
```
Implementation (2)

• 2\textsuperscript{nd} stage: linking
  – Final type checking
  – A reviser inherits from its target class
    • Revisers targeting the same class are linearized.
    • Object creation is modified.
      \[
      \text{new AddExpr()} \quad \rightarrow \quad \text{new StringEx()}
      \]
Implementation (3)

- within methods

```java
class L {
    ... 
    v = right.eval();
    ... 
}

v = right.eval_L();
```
Related work

• Predicate dispatch and AOP
  – D. Orleans pointed out in 2001
  – C. Bockisch [SPLAT’06], M. Haupt [ECOOP’07]

• Destructive extension
  – MixJuice, FOP, JavaGl, …
  – Open classes, expanders, …

• Limited scope
  – Classbox/J, COP, …
Not discussed AOP functionalities

• Pattern matching
  – A single advice may modify multiple methods matching a given pattern.

• dynamic pointcuts like cflow
Summary

• GluonJ
  – A simple AOP language with OOP-like constructs
    • Revisers and within methods
    • mostly modular
  – inherent in AOP?

• More in the paper
  – The type system is sound.
  – A formal model for compilation
    • GluonFJ to FJ translation
    • It preserves typing.
  – Benchmarking
    • Execution overheads are negligible.
  – http://www.javassist.org