Fine-Grained Generic Aspects

Tobias Rho, Günter Kniesel, Malte Appeltauer
Dept. of Computer Science III
University of Bonn
FOAL Workshop
Overview

• Motivation
• LogicAJ 2
• Examples
• Related Work
• Conclusion
Limitations of Common Aspect Languages

• Restricted set of join points
  ⇒ pointcuts just on the interface level (classes, methods, fields)
    ⇒ call, execution, set, get ...

⇒ fine-grained ?
  ⇒ e.g. loops ?

```
class Foo{
  private int myField;
  public void foo() {
    myField = 42;
    if (Bar.A < 4711) {
      myField = 1;
    } else {
      myField = 0;
    }
  }
  public int bar() {
    foo();
    return myField;
  }
  public void m(int[] a, int[] b) {
    for (int i = 0; i < a.length; i++) {
      a[i] = 2*i; b[i] = i*i;
    }
  }
}
```

```java
final Vector v = new Vector();
List list = new ArrayList();
final int range = (a.length - 0) / THREADS;
for (int i = 0; i < THREADS; i++) {
  final int finalThreads = i;
  Thread thread = new Thread() {
    public void run() {
      int newLb = lb + range * finalThreads;
      int newUB = newLb + range;
      if (newUB >= ub) newUB = ub;
      for (int i = newLb; i < newUB; i++) {
        a[i] = 2*i;
        b[i] = i*i;
      }
    }
  }.run();
  list.add(thread);
}
```
Limitations of Common Aspect Languages

• Restricted set of join points
  ⇒ pointcuts just on the interface level (classes, methods, fields)
    ⇒ call, execution, set, get ...
  ⇒ fine-grained?

AOP Test Coverage?

class Foo{
  private int myField;
  public void foo(){
      myField = 42;
      if(Bar.A < 4711){
          myField = 1;
      }
      else{
          myField = 0;
      }
  }
  public int bar(){
      foo();
      return myField;
  }
  public void (int[] a, int[] b) {
      for(int i = 0;i< a.length;i++) {
          a[i] = 2*i;
          b[i] = i*i;
      }
  }
}
Find Bad Smells

• Compile-time check for bad smells

```java
class Foo{
    public int getValue(){
        if (this instanceof A) return 42;
        if (this instanceof B) return 4711;
        else return 0;
    }
}
```

Bad Smell!

Compile-time warning: „Use Polymorphism!“

```java
aspect
myField.methFromKnownClass().methFromUnknownClass();

Bad Smell!

Compile-time warning: „Violates the Law of Demeter!“
```
LogicAJ 2

A fine-grained generic aspect language
Fine-grained pointcuts

• Fine-grained pointcuts
  ⇒ select all syntactically distinguishable join points
  → statements, expressions, declarations

  \[
  \begin{array}{l}
  \text{stmt}(\text{code}) \\
  \text{expr}(\text{code}) \\
  \text{decl}(\text{code})
  \end{array}
  \]

• Minimal language core
  ⇒ minimal number of pointcuts
  ⇒ simple, easy to learn language based on code patterns

  \[
  \text{pointcut} \ \text{fooBarCalls}(): \\
  \quad \text{expr}(\text{foo}()) \\
  \quad \| \| \quad \text{expr}(\text{bar}())
  \]

  ;
Extensible

• Extensible
  ⇒ build high-level pointcuts by composition
    → logic operations and recursion
  ⇒ e.g. static AspectJ pointcut semantics
  ⇒ loops, condition pointcuts

• Patterns?
  ⇒ Placeholders necessary!
  ⇒ Example task: Select all getter methods
    → First try: use wildcards

```java
pointcut getter():
    decl(public int get*(()) {
        *
        return *
    });
```
Use Logic Meta-Variables instead of Wildcards

• Transition from wildcards to meta-variables

```java
pointcut getter():
    decl( public int get*() {
        *
        return *;
    });

pointcut getter(?fname) :
    decl( public int ?getter() {
        ??stmts
        return ?fname;
    }) &&
    concat("get", ?fname, ?getter);
```
Logic Meta-Variables

⇒ Syntax: \(?lmv\)
⇒ Here: used to bind syntactically complete syntax elements
  → statements, expressions, declarations, (type) identifier
⇒ Special meta-variables for lists: \(??llmv\)

```java
pointcut getter(?fname) : 
  decl( public int ?getter() { 
    ??stmts 
    return ?fname;
  }) &&
  concat("get", ?fname, ?getter);
```
Relations between pointcuts

• Additional pointcut: setter

```java
pointcut setter(?fname):
    decl( public void ?setter(int ?param) {
        ??stmtbefore
        this.?fname = ?param;
        ??stmtafter
    }) &&
    concat("set", ?fname, ?setter);
```

• New contract: For every setter there is a getter method in the same class

```java
pointcut inconsistentGetterSetter():
    setter(?fname) &&
    !getter(?fname);
```

How do we express a relationship between selected join points?

INSUFFICIENT
Explicit join point variables

- Primitive pointcuts bind join points to meta-variables

\[
\begin{align*}
\text{stmt}(?jp, \text{code}) \\
\text{expr}(?jp, \text{code}) \\
\text{decl}(?jp, \text{code})
\end{align*}
\]
Explicit join point variables

- Primitive pointcuts bind the join points to a meta-variable

```java
pointcut getter(?jp, ?fname):
    decl ?jp, public int ?getter() {
        ?stmts
        return ?fname;
    }
} &&
concat("get", ?fname, ?getter);

pointcut setter(?jp):

pointcut inconsistentGetterSetter():
    setter(?setter, ?fname) &&
    !getter(?getter, ?fname);
```

How do we express a relationship between meta-variables?

STILL INSUFFICIENT
How do we ensure they are defined in the same class?
Finally …

- Use attributes to relate to meta-variable context

```java
pointcut getter(?jp):
    decl(?jp public int ?getter() {
        ??stmts
        return ?fname;
    }) &&
    concat(get, ?fname, ?getter);

pointcut setter(?jp): ...

pointcut inconsistentGetterSetter():
    setter(?setter, ?fname) &&
    !(getter(?getter, ?fname) &&
      equals(?getter::parent, ?setter::parent));
```
Logic Meta-Variable Attributes

- Meta-variable attributes provide context information
  ⇒ Syntax: \(?lmv::<attr>\)
- parent
  ⇒ The enclosing element
- ref
  ⇒ Resolved referenced declaration
- type
  ⇒ Resolved Java type of an element bound to a LMV
    (syntactic sugar, inferable via the ref attribute)
Generalized Aspect Construct

• Syntax

  explicit join point meta-variable

```
( introduce | before | after | around )
  <name>(<jp id>, <optional parameters>) : 
  <pointcut description>
{
  ( <class template> | 
  <method introduction> | 
  <field introduction> | 
  <advice body> )
}
```
LogicAJ 2 Summary

- Fine-grained aspect language
- Minimal set of basic pointcuts as a language core
- Uniform genericity
- Extensible
  - Build high-level pointcuts by composition
    - Logic operations and recursion
  - E.g. static AspectJ pointcut semantics
  - Loops, condition pointcuts
Constructing higher-level pointcuts
Named Pointcut Examples

- AspectJ call pointcut

```java
pointcut call(\$jp, \$mods, \$declType, \$returnType, \$name, \$parTypes):
    expr(\$jp, \$name(\$args))
    && decl(\$jp::ref, \$mods ?returnType ?name(\$par){\$stmts})
    && equals(?declType, ?jp::ref::parent::type)
    && parameterTypes(?parTypes, ?par);
```

select call expression and bind \$name to the method name and the LMV list variable \$args to the arguments list
Named Pointcut Examples

- AspectJ call pointcut

```java
pointcut call(?jp, ??mods, ?declType, ?returnType, ?name, ??parTypes):
  expr(?jp, ?name(??args) )
  && decl(?jp::ref, ??mods ?returnType ?name(??par){??stmts})
  && equals(?declType, ?jp::ref::parent::type)
  && parameterTypes(??parTypes, ??par);
```

select the syntax elements of the referenced method ?jp::ref
Using named pointcuts: Free Code Patterns

• Free Code patterns

```java
pointcut call(?jp, ??mods, ?declType, ?returnType, ?name, ??parTypes):
    ...
```

```java
after log(?jp):
    call(?jp, [public], ?ret, ClassX, ?m, [int] )
{
    System.out.println("called method" + ?m);
}
```

The call pointcut can easily be extended to support patterns.
For-loop pointcut

- For-loop

```java
  stmt(
      for(?jp, type ?index = ?lb; ?index < ?ub; ?incr) {
        ?statements
      }
  );
```
For-loop pointcut

- Using the For-loop pointcut

```java
?range::type around():
    forLoop(?range, ?lb, ?ub, ?incr, ??statements)
{
    < .. >
    int newLb = < .. >
    int newUb = < .. >
    for(?range::type ?range = newLb; ?range < newUb; ?incr) {
        ??statements
    }
    < .. >
}
```
Free Code Patterns

```plaintext
forLoop(?index, int, ?ub(), 0, ??statements)
```

bind meta-variables

```
free source patterns

int

?ub()
```

class C

```

m(int i)
```

for

```
init
cond
step
body
```

```
int i

<

i++
block

int

int i

m()
```
Related Work

• concrete solutions to a subset of join points
  ⇒ EOS (Hridesh Rajan et al.)
    → conditionals and loop pointcuts
  ⇒ LoopsAJ (B. Harbulot et al.)
    → loop pointcut, byte code analysis

• Extensible Compiler
  ⇒ abc Compiler (de Moor et al.)
    → aspect compiler framework
    → every part of the compiler is open to extension
    → still, for all extensions compiler knowledge is necessary

• JaTS
  ⇒ language for pattern based transformations of Java programs
  ⇒ code patterns describe program parts on which transformations should take place
  ⇒ transformation specification is described with another pattern
  ⇒ both parts can be linked by the use of meta-variables, which substitute syntactic elements at the interface level of a base-program
Conclusion

- fine-grained genericity for aspect languages
- base-language code patterns with meta-variables
- minimal set of fine-grained pointcuts
- express dependencies between multiple join points
- define arbitrary kinds of pointcuts that previously required specific language extensions
Questions?