Predicate dispatch for Aspect-Oriented Programming

Shigeru Chiba
Tokyo Institute of Technology
VMIL 2008

Understanding AOP

- What are primitive operations of AOP? – pointcut and advice?

- Primitives of OOP
  - dynamic method dispatch
  - instances
  - inheritance

OOP v.s. AOP

- Dynamic method dispatch
  - function calls are virtual.

- Virtual Join Points
  - [Bockish et al, SPLAT ’06]
  - [Haupt & Schippers, ECOOP ’07]
  - Join points are virtual
    - function calls, field accesses, ... and more

A way of overriding

- OOP
  - single dispatch, multi-dispatch, and predicate dispatch.

- AOP
  - more complex dispatch than predicate dispatch.
  - how?
This work

- What predicates do we need for AOP?
  - if we emulate AOP by predicate dispatch + open classes.

  - This would be beneficial for understanding AOP and the differences from OOP.

Predicate dispatch

- M. Ernst et al. [1998], Millstein [2004], ...
- Jpred
  - type, equality, linear arithmetic

```java
class TypeCheck {
    Type check(Env e, Node n) when n@IfStmtnt { ... }
    Type check(Env e, Node n) when n@WhileStmtnt { ... }
    Type check(Env e, Node n) when n == null { ... }
}
```

Logging aspect

- needs context-dependent dispatch.

```java
aspect Logging {
    around(): call(void HashMap.put(..)) && within(WebApp)) {
        System.out.println("Web app updates a hash map");
        proceed();
    }
}
```

Crosscutting concerns

- need to determine a method by global contexts.
  - within and withincde
    - e.g. call(* Rect.move(..)) && within(Window)
  - cflow
    - who is the caller?

  Jpred determines a method by only local contexts.
more predicates for AspectJ

• Predicates
  – `<var>` instanceof `<type>`
  – `<var>` statically-instanceof `<type>`
  – `<var>` running `<method>`

• Variables
  – this, parameters, `client`, `client*`
  (the if pointcut is not supported)

Drawbacks

• Jpred does not provide these predicates because JPred statically and modularly checks:
  – Exhaustiveness
    • no “message not understood” error happens during runtime.
  – No ambiguity
    • no “multiple methods are applicable” error happens during runtime.

Exhaustiveness

• No “message not understood” error
  – Open classes requires global analysis.
    • all aspects must be given at compile time.

  – `client/client*` (cflow) disable static check.
    • Or very conservative analysis
      (require the “base/default” method)

No Ambiguity

• No “multiple methods are applicable” error
  – Determine the most specific method.
  – JPred determines it by logical implication among predicates.
  – AspectJ determines it by the precedence relations among aspects.
A predicate representing precedence

- deployed(<aspect>)
  - deployed(A) is true if the aspect A is deployed.
  - deployed(A) implies deployed(B)
    if A has a higher precedence than B.

For normal methods,
- deployed(null) is always true
- deployed(A) always implies deployed(null)

Modularly checkable?

- Yes for JPred.
- No for AOP?
  - Only deploy(<aspect>) is used for determining
  - All aspects must be given at compile time.
  - client/client* (cflow) requires very conservative analysis.

Generic advices

- Pattern match
  - * (wildcard), + (subclasses), .. (any parameters)
  - e.g. javassist.CtClass+, Shape.set*(..)

- Solution
  - Multi-mixin [Apel 2005]
    - We can deal with an aspect as a template.
      A template instance is supplied to each class.
    - How to emulate proceed(), args(), ...??

Code example

- Class with pattern matching
  aspect Logging extends java.util.Map+ {
    require void toString();
    void put*(Object key, Object value)
    when within(WebApp) {
      System.out.println(toString());
      super.proceed(key, value);
    }
  }

- Predicates
  - target(java.util.Map+)
  - methodName(put*)
Aspect instances

- In AspectJ,
  - an aspect is instantiated. An advice is executed on that instance.

- Solution
  - Overriding methods can be forwarders to an aspect instance.

Summary

- AOP is as OOP with:
  - an extended method dispatching mechanism
    - global-contexts v.s. modular type check

- Future work
  - compare AOP with other language constructs
    - FOP, Classbox, eJava, JPred, ..
  - design a better AOP language
    - we can borrow ideas from other languages