Towards a Type System for Detecting Never-Matching Pointcut Compositions

Tomoyuki Aotani  Hidehiko Masuhara
Programming Principles and Practices Group
University of Tokyo
Never-matching pointcut

Don’t match any join point in any program

- `get(*)&&set(*)`
- `get(*)&&args(int)`

abstract aspect A{
  abstract pointcut p();
  after(int i):
    p()&&args(i){...}
}

aspect B extends A{
  pointcut p(): `get(*)`;
}

No get join point is a set join point
No get join point has an argument
Our approach: detect by using a type system

- Type of pointcuts
  - Represents attributes of matching join points
  - Is encoded by using record, union and the bottom types

- Guaranteed properties
  - Well-formedness of pointcuts
The property our type system assures: well-formedness

- Well-formed pointcuts:
  A pointcut $p$ is well-formed if there exists a well-typed program that has a join point matching $p$
Target language

- Subset of AspectJ’s pointcut language
  - \texttt{mget}(T C.f): selects a reference to an instance field (not declared as \texttt{static}).
  - \texttt{mset}(T C.f): selects an assignment to an instance field (not declared as \texttt{static}).
  - \texttt{args}(T1,\ldots,Tn): specifies the number of arguments and their types.
  - \texttt{p1} \&\& \texttt{p2}: makes an intersection of two pointcuts
  - \texttt{p1} \| \texttt{p2}: makes an union of two pointcuts
Typing rules for mget, mset and args pointcuts

• Assign record types that represent the properties of matching join points

- \text{mget}(T \ C.f) :
  \{\text{target} : C, \text{args} : \bullet, \text{kind} : \text{mget}, \text{name} : f, \text{ret} : T\}

- \text{mset}(T \ C.f) :
  \{\text{target} : C, \text{args} : [T], \text{kind} : \text{mset}, \text{name} : f, \text{ret} : \bullet\}

- \text{args}(T1,\ldots,Tn) : \{\text{args} : [T1,\ldots,Tn]\}

\text{T, C and f are identifiers or } * \text{.} \\
\bullet \text{ represents absence}
Typing rules for pointcut compositions

• $||$-composition

\[
\frac{pc_1 : P_1 \quad pc_2 : P_2}{pc_1 || pc_2 : P_1 + P_2}
\]

$P$: type of pointcut

$pc$: pointcut

$P$: union type

• $\&\&$-composition

\[
\frac{pc_1 : P_1 \quad pc_2 : P_2}{pc_1 \&\& pc_2 : P}
\]

$P < : P_1$

$P < : P_2$

$P$: a common subtype of $P_1$ and $P_2$
Type subsumption on the type of pointcuts

record subtyping

\{\text{args:}[C]\} \triangleright \{\text{args:}\bullet\}

\{\text{target: } *, \text{args:}[C], \text{kind: mget, ret: } C\} \triangleright \{\text{target: } C, \text{args:}[C], \text{kind: mget, ret: } \bullet\}

\{\text{target: } C, \text{args:}[C], \text{kind: mget, ret: } C\} \triangleright \{\text{args:}\bullet\}

never-matching pointcut

Any two types have a common subtype

bottom
Well-formed pointcut: \( \text{args(int)} && \text{mset(int Point.x)} \)

\[
\text{mset(int Point.x)}: \{ \text{target: Point, args: [int], kind: mset, name: x, ret: } \bullet \} \\
\text{args(int)}: \{ \text{args: [int]} \}
\]

\( T_1 \subseteq T_2 \quad T_1 \subseteq T_3 \)

\[
\text{args(int)} && \text{mset(int Point.x)}: \{ \text{target: Point, args: [int], kind: mset, name: x, ret: } \bullet \} 
\]
Never-matching pointcut:
`args(int) && mget(int Point.x)`

$mget(int Point.x)$:
{target: Point, `args: •`, kind:mget, name:x, ret:int}

$args(int)$: `{args: [int]}

have no common subtype

$bottom <: T_1$

$bottom <: T_2$

$args(int) && mset(int Point.x)$: $bottom$
Conclusion

• We defined the well-formedness of pointcuts
• We demonstrated our type system for pointcuts
  – The type of pointcuts is represented as record, union and the bottom types
  – Never-matching pointcuts have the bottom type
Future work

• Complete formalization
  – We use Featherweight Java [Igrashi01]
  – How can we define the typing rule for the `not (!)` pointcut?

• Prove type-soundness
  – Well-typed programs are well-typed after well-typed aspects are woven, and don’t go wrong

• Verify correctness of the design and implementation of pointcuts in AspectJ5
Related work (1/2):
Types and AspectJ-like AOPL

• **Typed parametric polymorphism for aspects**
  [Jagadeesan06]
  – provides AFGJ (FGJ [Igarashi01] + pointcut + advice + proceed)
    • join point: execution
    • pointcut: `exec`, `&&`, `||`
  – provides checking rules for pointcuts, which can successfully reject `exec(R C.m()) && exec(* C.m'( ))`

• **MiniMAO$_1$: An imperative core language for studying Aspect-Oriented reasoning**
  [Clifton06]
  – Classic Java [Flatt99] + aspect + pointcut + advice + proceed
    • join point: call, execution
    • pointcut: `call`, `exec`, `this`, `args`, `target`, `&&`, `||`, `!!`
  – provides typing rules for pointcuts but it cannot reject `exec(R C.m()) && call(R C.m())`
Related work (2/2): Types and Pointcuts

- **A Static Aspect Language for Checking Design Rules** [Morgan07]
  - develops a DSL that can be seen that enriches declare error/warning in AspectJ.
  - provides a type system that assures a pointcut matches at least one join point.
    - The typing rule for not pointcut is also defined.

- **A pointcut language for control-flow** [Douence04]
  - provides a richer pointcut language for control-flow than AspectJ’s.
  - discusses erroneous pointcut compositions and aspect interactions based on sets of join point shadows.
Typing rule for not pointcut in DSL\textsuperscript{[Morgan07]}

- $!pc$ has the same type to $pc$ i.e. $pc : P$
  $!pc : P$

<table>
<thead>
<tr>
<th>Joinpoint Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
<td>Namespace</td>
</tr>
<tr>
<td>type</td>
<td>Type</td>
</tr>
<tr>
<td>method</td>
<td>Method (including constructors)</td>
</tr>
<tr>
<td>argument</td>
<td>Method argument</td>
</tr>
<tr>
<td>field</td>
<td>Field</td>
</tr>
<tr>
<td>property</td>
<td>Property</td>
</tr>
<tr>
<td>event</td>
<td>Event</td>
</tr>
<tr>
<td>attribute</td>
<td>Attribute of a program element</td>
</tr>
<tr>
<td>genericArgument</td>
<td>Type argument (to a generic type)</td>
</tr>
<tr>
<td>bytecode</td>
<td>Instruction in the program</td>
</tr>
</tbody>
</table>
Well-formed pointcut: 
\((\text{mget}(* *.*)) \lor \text{mset}(\text{* *.*})) \land \text{args}(\text{int})\)

- \(\text{mget}(\text{* *.*}) \land \text{args}(\text{int}): \text{bottom}\)
- \(\text{mset}(\text{* *.*}) \land \text{args}(\text{int}): \{\text{target:*}, \text{args: [int]}, \text{kind:mset, name:*}, \text{ret:.}\}\)

Using the typing rule for \(\lor\)-compositions,

- \((\text{mget}(\text{* *.*}) \lor \text{mset}(\text{* *.*})) \land \text{args}(\text{int}): \{\text{target:*}, \text{args: [int]}, \text{kind:mset, name:*}, \text{ret:.}\} + \text{bottom}\)
Limitation of current type system

• ArrayList <: Object cannot be accepted
  – Our type system does not know the relation of ArrayList and Object

• Possible solution: specifying a reliable class hierarchy $H$

\[
H \vdash \text{ArrayList} <: \text{Object}
\]

\[
H \vdash \{\text{this:ArrayList}\} <: \{\text{this:Object}\}
\]
Unsafe join point reflection

caching return values of method calls

```
aspect Memoize{
    Hashtable store;
    after(): call(* *()){
        Object key=tjp.getTarget();
        if(!store.containsKey(key))
            store.add(key.clone(), proceed());
        return store.get(key);
    }
}
```

returns null when tjp matches calls to class methods

throws NullPointerException
Rejecting unsafe join point reflection

```java
aspect Memoize{
    Hashtable store;
    after(): call(* *()){
        Maybe<Object> key = tjp.getTarget();
        if(!store.containsKey(key))
            store.add(key.clone(), proceed());
        return store.get(key);
    }
}
```