Introducing $abc$

- why we are building it -
A workbench for AOP research

- Extensions:
  - Parametric introductions
  - Symmetric class composition (Hyper/J, GenVoca)
  - Trace cuts
  - Dataflow pointcuts

- New static checks:
  - Pure aspects

- Optimisations:
  - Eliminate runtime overheads for cflow
  - Avoid closures in around

require extensible frontend
require analysis framework
Requirements

**Syntactic changes** should be easy
⇒ Generate parser from grammar

**Frontend** framework designed for extensibility:
- Changes to environment type
- Augment existing type objects
- Same override on many existing AST classes
- Override in middle of AST inheritance hierarchy

**Backend** framework:
- Clean intermediate representation
- Rich set of existing analyses
- Easy to plug in new transformations

*Polyglot:* extensible Java compiler

*Soot:* manipulate classfiles, 3-addr IR for analysis and transformation

*jfex + javacup:* parser generator
abc team

- Chris Allan - Oxford (tracecuts)
- Pavel Avgustinov - Oxford (test harness, privileged, lexer)
- Aske Simon Christensen - Århus (architecture, decl parents, pattern matcher)
- Laurie Hendren - McGill (grammar and scanner, initial weaver)
- Sascha Kuzins - Oxford (around weaver)
- Jennifer Lhoták - McGill (JavaToJimple, initial weaver)
- Ondrej Lhoták - McGill (initial weaver, soot class and method handling)
- Oege de Moor - Oxford (frontend, intertype weaver)
- Damien Sereni - Oxford (cflow optimisations)
- Ganesh Sittampalam - Oxford (pointcut matching, advice weaver, optimisations, class and method handling)
- Julian Tibble - Oxford (EAJ language extensions)
Suggested schedule

*Morning*: introductory talks about abc (why, what and how)

*Afternoon*: discuss
- difficult points in language design
- plans for future of AspectJ
- opportunities for ajc/abc collaboration
Plan for the morning

- Architecture (Aske Simon Christensen)
- Scanner and Parser (Laurie Hendren)
- Polyglot and frontend (Oege de Moor)
- Introduction to Soot (Ondrej Lhoták)
- Advice weaver (Ganesh Sittampalam)
- Around weaver (Sascha Kuzins)
- Language extensions (Julian Tibble)
- Current status and plans (Oege de Moor)
abc architecture
Goals

• Focus on extensibility and runtime performance
• Separation between frontend (Polyglot) and backend (Soot)
• Whole-program compiler
Basic design of the compiler

Source files → Frontend (Polyglot) → Weaver (Soot) → Woven code

Class files → Frontend (Polyglot) → Weaver (Soot) → Woven code

Java IR → Weaver (Soot) → Woven code

Aspect info → Weaver (Soot) → Woven code
Aspect Info

- Internal representation of all AspectJ-specific constructs
- All code is put into placeholder methods
- Name mangling for intertype and access
- Categories and real signatures of methods and fields
- Mapping between Polyglot and Soot types
Components of the compiler

Frontend
- Input Source Files
- Input Class Files
- Library Class Files
  - Scan & Parse
  - AspectJ AST
    - Semantic Checks
      - Checked AspectJ AST
        - Separator
          - Java AST
          - Aspect Info

Weaver
- Java AST
- Aspect Info
  - Class File Resolving
    - AST Resolving
      - Class Skeletons
        - Static Weaving
          - Woven Class Skeletons
            - Class File Jimplification
              - AST Jimplification
                - Jimple
                  - Static Weaving Fixup
                    - Specially Woven Jimple
                      - Advice Weaving
                        - Woven Jimple

Output
- Woven Jimple
  - Optimization
    - Optimized Jimple
      - Output
        - Output Class Files
The abc scanner and parser

Laurie Hendren and the abc team
Challenges

- Unambiguous LALR(1) grammar for the complete AspectJ language that is a natural extension of the Java grammar. (easy to understand and extend)
- Express as much of the language specification in the grammar as possible (for example, differentiate in the grammar where class pattern is required and where a general type pattern is allowed).
- Handle the different sublanguages and associated reserved words in a well-defined manner.
**abc Solution Overview**

- Jflex-based scanner that is built on top of Polyglot’s Java scanner.

- abc’s scanner uses state to distinguish between different scanning contexts.
  
  abc/src/abc/aspectj/parse/aspectj.flex

- LALR(1) grammar expressed as a clean extension to Polyglot’s base Java grammar (originally defined by Scott Ananian - JavaCup)
  
  abc/src/abc/aspectj/parse/java12.cup
  abc/src/abc/aspectj/parse/aspectj.ppg
Really three different sublanguages:
1. normal Java code
2. aspect declarations
3. pointcut definitions

Different sub-languages have different lexical structure, for example

```
if*.*1.Foo+.new(..)
```

**Java:** reserved("if"), op("*"), op("."), op("*"), float(1.0), id("Foo"), op("+")
reserved("new"), op("("), op("."), op("."), op("")

**Pointcut:** IdPat("if*"), op("."), IdPat("*1"), op("."), Id("Foo"), op("+")
reserved("new"), op("("), op("."), op("."), op("")

The abc scanner and parser – p.3/16
abc Scanner Uses States

- Scanner maintains a stack of states.
- New state is pushed when entry into lexical scope is detected, and the scanner is put into the new state.
- When the end of a lexical state is detected, state is popped from the stack and scanner put into the state now at the top of the stack.
- Four major states, each state has well-defined entry/exit points, and its own lexical structure, including specific reserved words defined for that state.
- A reserved word is easily associated to two different token types, based on current state of the scanner. For example, `if` can have two different token types, one for the regular `if` and one for the pointcut `if`.
Scanner States

Java: Default state, `aspect`, `privileged`, and `pointcut` are reserved words. This state is entered at `class` or `interface` and exited at matching `}`. (finding the matching `}` requires a nesting counter)

Aspect: Begins at the `aspect` keyword and ends at the end of the aspect declaration’s body. Has, in addition to above reserved words, `after`, `around`, `before`, `declare`, `issingleton`, `percfflow`, `percfflowbelow`, `pertarget`, `perthis`, `pointcut`, and `proceed`.
**Pointcut:** Four contexts in which pointcut expressions may be found:

- per clause: `pertarget ( ...... )`
- declare declaration: `declare ...... ;`
- body of a pointcut declaration: `pointcut ...... ;`
- header of an advice declaration: `after ...... {`

Reserved words in this state are only:

- `adviceexecution args, call, cflow, cflowbelow, error, execution, get, handler, if, initialization, parents, precedence, preinitialization, returning, set, soft, staticinitialization, target, this, throwing, warning, within and withincode.`
**PointcutIfExpr**: inside a pointcut, an if pointcut has a nested expression, same scanning state as Aspect, but state returns to pointcut state at terminating parenthesis.

```
..... if ( ..... ) ..... 
```
Defining a LALR(1) grammar as Polyglot ext.

1. Define new alternatives to existing rules in the polyglot Java grammar.
2. Define new grammar productions. (sometimes must accept a slightly too large language and then weed)
All new alternatives

\[ \langle \text{type}\_\text{declaration} \rangle ::= \langle \text{aspect}\_\text{declaration} \rangle \]

\[ \langle \text{class}\_\text{member}\_\text{declaration} \rangle ::= \langle \text{aspect}\_\text{declaration} \rangle \]
\| \langle \text{pointcut}\_\text{declaration} \rangle

\[ \langle \text{interface}\_\text{member}\_\text{declaration} \rangle ::= \langle \text{aspect}\_\text{declaration} \rangle \]
\| \langle \text{pointcut}\_\text{declaration} \rangle

\[ \langle \text{method}\_\text{invocation} \rangle ::= \text{'proceed'} \ '(\langle \text{argument}\_\text{list}\_\text{opt} \rangle)' \]
Adding alternatives in Polyglot

/* add the possibility of declaring an aspect to type_declaration */

extend type_declaratiion ::= 
    aspect_declaratiion:a
    { : RESULT = a; : }

;
New aspect-specific productions

aspect_declaration ::= 
    modifiers_opt:a PRIVILEGED modifiers_opt:a1 
    ASPECT:n IDENTIFIER:b 
    super_opt:c interfaces_opt:d 
    perclause_opt:f 
    aspect_body:g 
    { : RESULT = parser.nf.AspectDecl(parser.pos(n,g), 
        true, a.set(a1), b.getIdentifier(), 
        c, d, f, g); 
    :}
aspect_declaration (continued)

| modifiers_opt:a
    ASPECT:n IDENTIFIER:b
    super_opt:c interfaces_opt:d
    perclause_opt:f
    aspect_body:g

{: RESULT = parser.nf.AspectDecl(parser.pos(n,g),
    false, a, b.getIdentifier(),
    c, d, f, g);

:}
abc grammar includes pointcuts

\[
\langle \text{basic}\_\text{pointcut}\_\text{expr} \rangle ::= \\
\quad \left( \langle \text{pointcut}\_\text{expr} \rangle \right) \\
\quad \text{`call'} \left( \langle \text{method}\_\text{constructor}\_\text{pattern}\_\text{expr} \rangle \right) \\
\quad \text{`execution'} \left( \langle \text{method}\_\text{constructor}\_\text{pattern}\_\text{expr} \rangle \right) \\
\quad \text{`initialization'} \left( \langle \text{constructor}\_\text{pattern}\_\text{expr} \rangle \right) \\
\quad \text{`preinitialization'} \left( \langle \text{constructor}\_\text{pattern}\_\text{expr} \rangle \right) \\
\quad \text{`staticinitialization'} \left( \langle \text{classname}\_\text{pattern}\_\text{expr} \rangle \right) \\
\quad \text{`get'} \left( \langle \text{field}\_\text{pattern} \rangle \right) \\
\quad \text{`set'} \left( \langle \text{field}\_\text{pattern} \rangle \right) \\
\quad \text{`handler'} \left( \langle \text{classname}\_\text{pattern}\_\text{expr} \rangle \right) \\
\quad \ldots
\]
\langle \text{basic} \_\text{pointcut} \_\text{expr} \rangle ::= \ldots

| ’adviceexecution’ ’( ’ ’)’
| ’within’ ’( ’\langle \text{classname} \_\text{pattern} \_\text{expr} \rangle ’)’
| ’withincode’ ’( ’\langle \text{method} \_\text{constructor} \_\text{pattern} \rangle ’)’
| ’cflow’ ’( ’\langle \text{pointcut} \_\text{expr} \rangle ’)’
| ’cflowbelow’ ’( ’\langle \text{pointcut} \_\text{expr} \rangle ’)’
| ’if’ ’( ’\langle \text{expression} \rangle ’)’
| ’this’ ’( ’\langle \text{id} \_\text{star} \rangle ’)’
| ’target’ ’( ’\langle \text{id} \_\text{star} \rangle ’)’
| ’args’ ’( ’\langle \text{id} \_\text{star} \_\text{list} \_\text{opt} \rangle ’)’
| \langle \text{name} \rangle ’( ’\langle \text{id} \_\text{star} \_\text{list} \_\text{opt} \rangle ’)’
Specific Patterns

\[
\langle \text{method	extunderscore constructor	extunderscore pattern} \rangle ::= \\
\langle \text{method	extunderscore pattern} \rangle \\
| \langle \text{constructor	extunderscore pattern} \rangle \\
\]

\[
\langle \text{method	extunderscore pattern} \rangle ::= \\
\langle \text{modifier	extunderscore pattern	extunderscore expr} \rangle \langle \text{type	extunderscore pattern	extunderscore expr} \rangle \\
\langle \text{classtype	extunderscore dot	extunderscore id} \rangle \\
\langle \text{throws	extunderscore pattern	extunderscore list	extunderscore opt} \rangle \\
| \langle \text{type	extunderscore pattern	extunderscore expr} \rangle \langle \text{classtype	extunderscore dot	extunderscore id} \rangle \\
\langle \text{throws	extunderscore pattern	extunderscore list	extunderscore opt} \rangle \\
\]

\[
\langle \text{method	extunderscore pattern} \rangle ::= \\
\langle \text{modifier	extunderscore pattern	extunderscore expr} \rangle \langle \text{type	extunderscore pattern	extunderscore expr} \rangle \\
\langle \text{classtype	extunderscore dot	extunderscore id} \rangle \\
\langle \text{throws	extunderscore pattern	extunderscore list	extunderscore opt} \rangle \\
| \langle \text{type	extunderscore pattern	extunderscore expr} \rangle \langle \text{classtype	extunderscore dot	extunderscore id} \rangle \\
\langle \text{throws	extunderscore pattern	extunderscore list	extunderscore opt} \rangle \\
\]
Summing up ....

- State-based scanner, plus LALR(1) grammar:
  - clearly defines lexical scopes and associated reserved words
  - naturally handles different sub-languages in AspectJ
  - clean addition to the base Java grammar
  - easy to understand
  - easy to extend

- More detailed scanning/parsing document at:
  http://abc.comlab.ox.ac.uk/doc
AspectJ as a Polyglot extension

- the frontend of abc -
Roadmap

- What is Polyglot?
- Brief overview of the AspectJ extension
- Sketch of disambiguation of “this” in ITDs
- Summary
What is Polyglot?

An extensible Java compiler

Sample extensions:

- Jif: Java information flow and program partitioning
- PolyJ 2.0: Java with parameterized types
- JMatch: Abstract iterable pattern matching for Java
- Jx: Nested inheritance in Java
- Jedd: BDD-based analyses
- JPred: Practical predicate dispatch

Produced by Andrew Myers, Nate Nystrom et al. at Cornell
How does Polyglot do it?

• Structured as a series of visitors
• Each visitor pass rewrites AST; about 15 such visitors
• Rigorous use of interfaces and factories makes it easy to change type system, environment, ...
• Delegates for overriding members of non-final AST classes (cf. intertype decls)
The AspectJ extension

Like any other Polyglot extension, five new packages:

- **AST**: new ast nodes (89 classes)
- **Extension**: overrides of existing Java AST nodes (13 classes)
- **Parse**: new lexer and grammar (2 files)
- **Types**: new types and type system (8 classes)
- **Visit**: new passes (35 classes)

- Includes Java/AspectInfo separator
- Many AST classes in pointcut language are light-weight
- The tricky bits are the type rules for ITDs, and the separator into Java & AspectInfo
Example: intertype scope rules

```java
public class A {
    int x;
    class B { int x; }
}

aspect Aspect {
    static int x;
    static int y;
    int A.B.foo() {
        class C {
            int x = 3;
            int bar() { return x + A.this.x; }
        }
        return this.x + (new C()).bar() + y;
    }
}
```
Example: intertype scope rules

```
public class A {
    int x;
    class B { int x; }
}
aspect Aspect {
    static int x;
    static int y;
    int A.B.foo() {
        class C {
            int x = 3;
            int bar() {return x + A.this.x;}
        }
        return this.x + (new C()).bar() + y;
    }
}
```

need to disambiguate field references:
- may be a reference to aspect fields,
- local class fields,
- or host (=target) of intertype declaration

Rules:
- no explicit receiver? if it was introduced into environment by the host, give it “this” from host.
- explicit “this” or “super”? if there is no qualifier and we're not inside a local class, it refers to the host. If there is a qualifier Q, and there is no enclosing instance of type Q nested inside the ITD, it refers to the host if the host has an enclosing instance of type Q.
How to disambiguate “this”

- Extend context type in Polyglot
- Test to determine whether this refers to host
- Override disambiguate for Polyglot this.
New context type

types.Context:

```java
public interface AJContext extends Context {
    Context pushHost(ClassType ct, boolean declaredStatic);
        // called when entering itd
    ClassType hostClass();  // return target of current itds
    boolean inInterType();  // are we inside an intertype declaration?
    boolean nested();        // are we inside a local class in an intertype declaration?

    // other itd-related members...
    boolean varInHost(String name);
    boolean methodInHost(String name);
    ClassType findFieldScopeInHost(String name);
    ClassType findMethodScopeInHost(String name) throws SemanticException;
    // ... more for advice and declare decls ...
}
```
Does “this” refer to host of ITD?

types.AJTypeSystem_c

```java
public boolean refHostOfITD(AJContext c, Typed qualifier) {
    if (!c.inInterType()) // if not inside an ITD, cannot refer to a host
        return false;
    if (qualifier == null) // if there is no qualifier
        return !c.nested(); // it refers to the host if we're not in a local class
    else // otherwise look for enclosing instance in host
        return c.hostClass().hasEnclosingInstance(qualifier.type().toClass());
}
```
Override disambiguate

extension.AJSpecial_c (Special is the Polyglot class to represent “this”):

```java
public Node disambiguate(AmbiguityRemover ar) throws SemanticException {
    AJContext c = (AJContext) ar.context();
    AJTypeSystem ts = (AJTypeSystem) ar.typeSystem();
    if (!(ts.refHostOfITD(c, qualifier())))) {
        // this is an ordinary special, it does not refer to the host
        return super.disambiguate(ar);
    } else {
        // this is a host special
        AJNodeFactory nf = (AJNodeFactory) ar.nodeFactory();
        HostSpecial_c hs = (HostSpecial_c) nf.hostSpecial(position, kind, qualifier, ((AJContext)c).hostClass());
        return hs.type(type()).disambiguate(ar);
    }
}
```
Frontend summary

✓ Extensible in all dimensions:
  - syntax, type system, visitors
✓ Potential merge problems with pure Java compiler only occur in extension dir and type system
✓ Extensions to abc have same structure as abc itself
Soot, a Tool for Analyzing and Transforming Java Bytecode

presenter: Ondřej Lhoták, McGill University
Soot toolkit

Soot provides:

- Convenient IRs (mainly Jimple)
- Existing analyses and transformations
- Framework for new analyses, transformations, code generation
- Dava decompiler
- Eclipse plugin for visualization
- Whole-program analysis framework
Soot Overview

Soot

Bytecode-to-Jimple Front-end

Java-to-Jimple Front-end

.jimple

Analyses and Transformations

Add tags

Generate Bytecode

Decompile to Java

.class

.java

Polyglot
Jimple

Jimple is:

- principal Soot Intermediate Representation
- 3-address code in a *control-flow graph*
- a *typed* intermediate representation
- *stackless*
**Jimple example**

**Java:**
```java
public int bar(int a, int b) {
    return a+b;
}
```

**Jimple:**
```java
public int bar(int, int) {
    Foo this;
    int a, b, $i0;

    this := @this;
    a := @parameter0;
    b := @parameter1;
    $i0 = a + b;
    return $i0;
}
```
Converting bytecode $\rightarrow$ Jimple $\rightarrow$ bytecode

- These transformations are relatively hard to design so that they produce correct, useful and efficient code.
- Worth the price, we do want a 3-addr typed IR.

<table>
<thead>
<tr>
<th>raw bytecode</th>
<th>typed 3-address code (Jimple)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- each inst has implicit effect on stack</td>
<td>- each stmt acts explicitly on named variables</td>
</tr>
<tr>
<td>- no types for local variables</td>
<td>- types for each local variable</td>
</tr>
<tr>
<td>- $&gt; 200$ kinds of insts</td>
<td>- only 15 kinds of stmts</td>
</tr>
</tbody>
</table>
Bytecode $\rightarrow$ Jimple

- Naive translation from bytecode to untyped Jimple, using variables for stack locations.
- splits DU-UD webs (so many different uses of the stack do not interfere)
- types locals (SAS 2000)
- cleans up Jimple
Java → Jimple

- Input: Polyglot AST generated from .java sources
- Compile AST to Jimple
- Generate Jimple methods/classes for implicit Java features (initializers, inner class accessor methods, class literals, assertions)
- Output: Jimple to be analyzed/optimized, eventually converted to bytecode
- Combination of Polyglot, Java-to-Jimple, Jimple-to-Bytecode passes forms a complete Java compiler equivalent to javac.

This part is unchanged in abc.
A naive translation introduces many spurious stores and loads.

Two approaches (CC 2000),
- aggregate expressions and then generate stack code; or
- perform store-load and store-load-load elimination on the naive stack code.
public class Foo {
    public int foo(int x, int y, int z) {
        return bar(x, y, z);
    }
    public int bar(int a, int b, int c) {
        return a+b+c;
    }
}

aspect A {
    before(Foo x) :
        call(int bar(int, int, int)) && target(x) {
            System.out.println(x);
        }
}
public int foo(int x, int y, int z)
0:     aload_0
1:     iload_1
2:     iload_2
3:     iload_3
4:     invokevirtual Foo.bar (III)I (7)
7:    IRETURN
Weaving example – weaving by hand

```java
public int foo(int x, int y, int z)
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>invokestatic A.aspectOf ()LA;</td>
<td>(14)</td>
</tr>
<tr>
<td>3:</td>
<td>aload_0</td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>invokevirtual A.before$0 (LfFoo;)V</td>
<td>(20)</td>
</tr>
<tr>
<td>7:</td>
<td>aload_0</td>
<td></td>
</tr>
<tr>
<td>8:</td>
<td>iload_1</td>
<td></td>
</tr>
<tr>
<td>9:</td>
<td>iload_2</td>
<td></td>
</tr>
<tr>
<td>10:</td>
<td>iload_3</td>
<td></td>
</tr>
<tr>
<td>11:</td>
<td>invokevirtual Foo.bar (III)I</td>
<td>(9)</td>
</tr>
<tr>
<td>14:</td>
<td>ireturn</td>
<td></td>
</tr>
</tbody>
</table>
Weaving example – ajc weaving

```java
public int foo(int x, int y, int z)
    0:  aload_0
    1:  iload_1
    2:  iload_2
    3:  iload_3
    4:  istore %4
    6:  istore %5
    8:  istore %6
   10: astore %7
   12: invokestatic A.aspectOf ()LA; (52)
   15: aload %7
   17: invokevirtual A.ajc$before$A$124 (LFoo;)V (56)
   20: aload %7
   22: iload %6
   24: iload %5
   26: iload %4
   28: invokevirtual Foo.bar (III)I (37)
   31: ireturn
```
public int foo(int, int, int)
{
    Foo this;
    int x, y, z, $i0;

    this := @this;
    x := @parameter0;
    y := @parameter1;
    z := @parameter2;
    $i0 = this.bar(x, y, z);
    return $i0;
}
public int foo(int, int, int)
{
    Foo this;
    int x, y, z, $i0;
    A theAspect;

    this := @this;
    x := @parameter0;
    y := @parameter1;
    z := @parameter2;
    theAspect = A.aspectOf();
    theAspect.before$0(this);
    $i0 = this.bar(x, y, z);
    return $i0;
}
Weaving example – bytecode from Jimple

public int foo(int x, int y, int z)
0: invokestatic A.aspectOf ()LA; (14)
3: aload_0
4: invokevirtual A.before$0 (LFoo;)V (20)
7: aload_0
8: iload_1
9: iload_2
10: iload_3
11: invokevirtual Foo.bar (III)I (9)
14: ireturn
Intraprocedural analyses and transformations

- local packer ("register allocation" on bytecode locals)
- copy propagation
- constant propagation
- common subexpression elimination
- partial redundancy elimination
- dead assignment elimination
- unreachable code elimination
- branch simplification
Law of Demeter benchmark

In method
lawOfDemeter.objectform.Pertarget.fieldIdentity:

- ajc: 616 locals
- ajc+Soot: 3 locals
- abc: 3 locals
Law of Demeter benchmark

In method
lawOfDemeter.objectform.Pertarget.fieldIdentity:

- ajc: 616 locals 45.9 seconds
- ajc+Soot: 3 locals 14.1 seconds
- abc: 3 locals 1.0 second
Adding analyses and transformations

Soot provides tools:
- control flow graphs
- def/use relationships
- fixed-point flow analysis framework
- method inliner

These are useful to have available for:
- weaving itself
- optimizing woven code
Dava decompiler

```java
public int foo(int x, int y, int z) {
    A.aspectOf().before$0(this);
    return this.bar(x, y, z);
}
```

- Dava decompiles bytecode with strange aspect-generated control flow that breaks other decompilers.
- Dava is integrated with Soot and abc. We could produce annotated decompiled output (e.g. comments showing pointcuts).

Soot, a Tool for Analyzing and Transforming Java Bytecode – p.19
Eclipse plugin

- Soot can be run as a plugin from Eclipse.
- Soot includes a tagging framework to communicate analysis information to Eclipse for visualization. (CC2004, eTX2004)
- Could be used to communicate aspect-specific information.
Whole-program analyses

- CHA call graph
- VTA – more precise call graph (OOPSLA2000)
- Spark: context-ins. points-to and call graph (CC2003)
- Paddle: BDD based framework for context-sensitive:
  - points-to analysis
  - call graph analysis
  - cflow analysis
  - type analysis (instanceof checks)
  - side-effect analysis (aspect purity)
  - escape analysis (thisJoinPoint[StaticPart])
Advice weaving

Ganesh Sittampalam
Overview

• Match - produce mapping:
  application sites ➔ advice + dynamic residue

• Prepare application sites

• Weave “inside-out” (i.e. in reverse precedence order)
Pointcut separation

- Restrict containing class
  - e.g. within(...)  
  - Does include nested classes
- Restrict containing method
  - e.g. withincode(...)  
  - Doesn’t include classes lexically within the method
- Specific join point
  - e.g. call(...)
Translating pointcuts

- `execution(int Foo.foo(char))` ➜ `withinmethod(int Foo.foo(char)) && execution()`
- `execution(Foo.new(int))` ➜ `withinconstructor(Foo.new(int)) && execution()`
- `adviceexecution()` ➜ `withinadvice() && execution()`
- `staticinitialization(Foo)` ➜ `within(Foo) && withinstaticinitialization() && execution()`
- `preinitialization(Foo.new(int))` ➜ `withinconstructor(Foo.new(int)) && preinitialization()`
- `call(int Foo.foo(char))` ➜ `methodcall(int Foo.foo(char))`
- `call(Foo.new(int))` ➜ `constructorcall(Foo.new(int))`
Initialization

initialization(Foo.new(int))
  ➜ withinconstructor(Foo.new(int))
  && classinitialization()

initialization(Foo.new())
  ➜ (withinconstructor(Foo.new())
      && classinitialization()))
  || interfaceinitialization(Foo)

initialization(Foo.new(..))
  ➜ (withinconstructor(Foo.new(..))
      && classinitialization()))
  || interfaceinitialization(Foo)
Pointcut preprocessing

- Inline named pointcuts
  - requires “private” pointcut variables
    ```java
    pointcut bar(int x) : args(x,..)
    bar(*) => private(int x) { args(x,..) }
    ```

- Convert to DNF
  - to correctly handle alternative bindings
    ```java
    (this(x) || target(x)) && if(x instanceof Foo)
    => (this(x) && if(…)) || (target(x) && if(…))
    ```

- Lift pointcuts from cflow and per clauses into special advice declarations
  - look for CSE and counter opportunities with cflow pointcuts
Restructuring

- **Move** `new+invokespecial` together
  - Needed for constructor call matching
- `foo()` → `a0 = foo()`
  - If `foo()` returns a value we want to bind
- **Restructure** `return` statements in body so that there is just one at the end
  - For execution pointcuts
- **Inline** `this(...)` calls in constructors
  - For initialization and preinitialization weaving
Matching

• Shadows categorised as:
  – Whole body (execution, initialization etc)
  – Individual statement (method call, field set, field get etc)
  – Pair of statements (constructor call)
  – Exception handler

• Iterate through all weavable classes
  – At each shadow, try all pointcuts
Finding method call shadows

... if (stmt instanceof InvokeStmt) {
    InvokeStmt istmt=(InvokeStmt) stmt;
    invoke=istmt.getInvokeExpr();
} else if (stmt instanceof AssignStmt) {
    AssignStmt as = (AssignStmt) stmt;
    Value rhs = as.getRightOp();
    if(!((rhs instanceof InvokeExpr)) return null;
    invoke=(InvokeExpr) rhs;
} else return null;

SootMethodRef methodref=invoke.getMethodRef();
Dynamic residues

- Mini-language roughly corresponding to structure of pointcuts
- Used to generate runtime code
  - decide whether advice should execute
  - bind values to pass to advice
- Also used to signal static results
  - “Match failed”
  - “This always matches”
- Easy to improve residues using analysis results
Dynamic residue construction

- “pre” residue from aspect
  - hasAspect check for per advice
- Residue from pointcut
- Residue from advice spec (before, after etc)
- “post” residue from aspect
  - aspectOf for getting aspect instance
Weaving

• Insert nops around the instruction(s) representing the shadow
  – Take care to fix up exception ranges and gotos correctly
• Advice gets inserted just inside the nops
• Advice gets woven “inside-out”
Around Weaving in abc
Objectives

• Avoid heap allocations
• Inlining not as the general strategy
  – to avoid code duplication
• Keep code in original classes
  – to avoid visibility problems
The starting point

• Around advice → advice method
  – same return type
  – arguments matching the advice formals
    • plus arguments for thisJoinpoint etc.

• proceed statement → call to dummy method

• Dynamic residue AST
  – includes all the bindings
  – (can fail)
Review: Closure strategy

• closure interface:

```java
public interface AroundClosure$1 {
    public [ret-type] proceed([arg-type] arg1, ...);
}
```

• advice method:

```java
[ret-type] adviceMethod$1(AroundClosure$1 closure,
                          [arg-type] arg1, ...) {
    ...
    [ret-type] result=closure.proceed(arg1', ...);
    ...
    return result;
}
```
Review: Closure strategy (2)

- Closure instantiation

```java
public class ShadowClass {
    public void shadowMethod() {
        AroundClosure$1 closure=new AroundClosure$1$Implementation$1();
        ...store additional information...
        Aspect.aspectOf().adviceMethod$1(closure, arg1, ...);
    }
    ...
}
```

- Closure implementation

```java
public class AroundClosure$1$Implementation$1 implements AroundClosure$1 {
    public [ret-type] proceed([arg-type] arg1, ...) {
        ... do what the shadow did...
    }
}
```
Avoiding the closure (1)

• Using the object itself
  – simply add an interface to the class of the shadow

```java
public class ShadowClass implements AroundClosure$1 {
    public [ret-type] proceed([arg-type] arg1, ...) {
        ...do what the shadow did...
    }
    public void shadowMethod() {
        Aspect.aspectOf().adviceMethod$1(this, arg1, ...);
    }
}
```
Avoiding the closure (2)

• Problem: The same advice can apply multiple times within the same class
• Solution: the shadow ID
public class ShadowClass implements AroundClosure$1 {
    public \[ret-type\] proceed(int shadowID, \[arg-type\] arg1, ...) {
        switch (shadowID) {
            case 0:
                ... do what the first shadow did...
            case 1:
                ... do what the second shadow did...
        }
    }
    public void shadowMethod() {
        Aspect.aspectOf().adviceMethod$1(this, 0, arg1, ...);
    }
    public void anotherShadowMethod() {
        Aspect.aspectOf().adviceMethod$1(this, 1, arg1, ...);
    }
}
Shadow ID (2)

• Problem: inheritance
  – subclasses may need to implement the same interface, but this overrides the original implementation of the superclass

• Solution: unique shadow ID, super() call
public class ShadowClassExt extends ShadowClass
    implements AroundClosure$1 {
    public [ret-type] proceed(int shadowID, [arg-type] arg1, ...) {
        switch (shadowID) {
            case 2:
                ... do what the shadow did...
                break;
            default:
                super(shadowID, arg1, ...);
        }
    }
    public void anotherShadowMethod() {
        Aspect.aspectOf().adviceMethod$1(this, 2, arg1, ...);
    }
}
Static methods

• Problem: shadows in static methods.
  – which object instance do we pass as the closure?
  – ideas:
    • create a temporary instance
    • use a singleton instance
Static Class ID

• Solution: the static class ID.
  – assign a unique integer ID to each class
  – implement a static proceed method where necessary.
  – pass this ID to the advice method
  – transform each proceed call into a switch statement
Static Class ID (2)

• static proceed method, unique id

```java
public class ShadowClass implements AroundClosure$1 {
    public static [ret-type] proceed_s(int shadowID,
                                        [arg-type] arg1, ...) {
        switch (shadowID) ... as before ...
    }
    public static shadowMethod() {
        Aspect.aspectOf().adviceMethod$1(null, 0,
                                          1, arg1, ...);
    }
}
```
Static Class ID (3)

```java
[ret-type] adviceMethod$1(AroundClosure$1 closure, int shadowID,
    [arg-type] arg1, ...) {
    ...
    closure.proceed(shadowID, arg1, ...);
    ...
}

[ret-type] adviceMethod$1(AroundClosure$1 closure, int shadowID,
    int staticClassID, [arg-type] arg1, ...) {
    ...
    switch (staticClassID) {
    case 0: closure.proceed(shadowID, arg1, ...); break;
    case 1: ShadowClass.proceed_s(shadowID, arg1, ...); break;
    ...
    }
    ...
```
Static Class ID (4)

- This method for the static cases can also be used for the non-static cases
- Tests indicate that this method is slightly faster
Transferring joinpoint context

- abc adds arguments to the advice method and the proceed method to carry the context
  - no heap allocations
- Problem: advice can apply to different joinpoints with different context
- Solution: add enough arguments to handle all the cases
Transferring joinpoint context (2)

- Mapping types
  - all reference types: Object
  - simple types are mapped to themselves
    - int-like types (short, byte, boolean and char) are mapped to int
  - (possibility of using exact reference types to avoid casts)
- This approach does not need boxing/unboxing for simple types
public class Foo {
    public static void main(String args[]) {
        new Foo().bar1("test");
        new Foo().bar2(1.0d);
    }
    public void bar1(String s) {}
    public void bar2(double d) {}
}

aspect Aspect {
    void around(): call(void * .bar*(..)) {
        proceed();
    }
}
public class Foo {
    public static void proceed$1(int shadowID,
        java.lang.Object contextArg1,
        double contextArg2,
        java.lang.Object contextArg3) {

        switch (shadowID) {
            case 0: ((Foo) contextArg1).test2(contextArg2);
                return;
            case 1: ((Foo) contextArg1).test1(contextArg3);
                return;
            default: throw new RuntimeException();
        }
    }

    public static void main(java.lang.String[] r0) {
        Foo target1 = new Foo();
        Aspect.aspectOf().adviceMethod$1(1, 1, target1, 0.0, "test");
        Foo target2 = new Foo();
        Aspect.aspectOf().adviceMethod$1(0, 1, target2, 1.0, null);
        return;
    }
}...
Transferring joinpoint context (5)

```java
class Aspect {
    final void adviceMethod$1(int shadowID,
                              java.lang.Object contextArg1,
                              double contextArg2,
                              java.lang.Object contextArg3)
    {
        ... Foo.proceed$1(shadowID,
                         contextArg1,
                         contextArg2,
                         contextArg3);
        ...
        return;
    }
    ...
}
```
Binding context

- When skipping the advice, the advice formals must be ignored
- The Skip Flag indicates this to the proceed method
• Example program

```java
public class Foo {
    public static void main(String args[]) {
        new Foo().bar(0);
    }
    public void bar(int i) {}
}

class Aspect {
    void around(int intArg):
        call(void *.bar*(..)) &&
        args(intArg) &&
        target(Foo)
        { proceed(intArg);
        }
}
```
public class Foo {
    public static void proceed$0(
        int intArg, // advice formal
        int shadowID, boolean skipFlag,
        java.lang.Object contextArg1, int contextArg2 ) {

        int arg;
        switch(shadowID) {
            case 0:
                if (skipFlag)
                    arg=contextArg2; // unbound case
                else
                    arg=intArg;      // bound case
                    
                    Foo callTarget=(Foo)contextArg1; // never bound
                    callTarget.bar(arg);
                    break;
            default: throw new RuntimeException();
        }
    }
}
```java
public class Foo {
    ...
    public static void main(String args[]) {
        Foo foo=new Foo();
        int i=0;
        if (foo instanceof Foo) {
            // residue passed
            Aspect.aspectOf().adviceMethod$0(...);
        } else {
            // residue failed
            proceed$0(......
                true, // skip flag
                ...);
        }
    }

    public void bar(int i) {}  
}
```
Alternative bindings

```java
aspect Aspect {
    void around(String s): call(void *.foo*(..)) &&
    (args(s,..) || args(.., s))
    {
        proceed("new");
    }
}
public class Foo {
    public static void main(String args[]) {
        new Foo().foo("string", new Integer(0));
        new Foo().foo(new Integer(0), "string");
    }
    public void foo(Object ob1, Object ob2) {
        System.out.println(ob1 + "", "" + ob2);
    }
}
```

Output:
new, 0
0, new
public class Foo {
    public static void main(String args[]){
        Foo foo=new Foo();
        Object arg1="string";
        Object arg2=null;
        String adviceFormal;
        int bindMask=0; // initialization
        label_0: {
            if (arg1 instanceof String) {
                adviceFormal=arg1;
                bindMask|=0; // removed by optimizer
            } else {
                if (arg2 instanceof String) {
                    adviceFormal=arg2;
                    bindMask|=2; // set bit 1
                } else { // skipped case
                    bindMask=1; // set skip flag
                    adviceFormal=null;
                    proceed_s$0(adviceFormal, 0, bindMask, foo, arg1, arg2);
                }
            }
            Aspect.aspectOf().adviceMethod$0(
                adviceFormal, null, 0, 1, bindMask, foo, arg1, arg2);
        }
    }
}
public class Foo {
    public static void proceed_s$0(String s, int shadowID, int bindMask,
            Object contextArg1, Object contextArg2, Object contextArg3) {
        ...Object arg1; Object arg2;
        if (bindMask==1) { // skip case
            arg1=contextArg2;
            arg2=contextArg3;
        } else {
            arg1=contextArg2; // first assign the default context
            arg2=contextArg3;
            switch ((bindMask & 2) >> 1) { // then overwrite the bound value
                case 0: arg1 = s; break;
                case 1: arg2 = s; break;
                default: throw new RuntimeException();
            }
        }
        Foo foo(Foo)contextArg1; // never bound
        foo.foo(arg1, arg2);
        ...
    }
}
Local and anonymous classes

• Problem: *proceed* in local/anonymous classes
  – can occur at an arbitrarily deep nesting level

• Solution: All relevant parameters of the advice method are stored as dedicated fields in each class at the outermost nesting level

• Classes at a deeper nesting level refer to the enclosing outermost class
Advice execution

- Around-advice applying to the execution of around-advice
- Weaving is done as described
- Problem: once an advice method has been woven into, it itself cannot be woven anymore
- Solution: topological sort of graph of applications
Circular advice execution

• Detected by topological sort
• Once an advice method has been woven into, use closure approach
  – closure simply implements interface of that advice method
• Closures or similar construct necessary
Closures

• Dedicated fields for all values
  – no Object array
• Actual shadow is moved to static method inside of original class
• No closure creation if residue fails
aspect Nullptr {
    pointcut methodsThatReturnObjects():
        ...
    Object around():
        methodsThatReturnObjects()
        Object lRetVal = proceed();
        if (lRetVal == null)
            System.err.println("Null return value: " + thisJoinPoint);
        return lRetVal;
    }
    A: pointcut methodsThatReturnObjects():
        call(* *.*(..)) && !call(void *.*(..));
    B: pointcut methodsThatReturnObjects():
        call(Object+ *.*(..));
    C: pointcut methodsThatReturnObjects():
        call(Object+ *.*(..)) && !within(lib.aspects..*);
Benchmarks – Nullptr (2)
Benchmarks (2) - Closures

![Bar chart showing benchmarks for A, B, C, and Base]
Future work

• Obvious optimizations
  – unused arguments, conditionals, table-switch etc.

• Adaptive inlining
  – post processing step

• Optimization of advice execution cycles
  – reduce likelihood of closure creation
Extending abc
Aspect Bench Compiler

• abc...
  – ...is designed to provide a workbed for research and investigation
  – ...therefore must be flexible and extensible

• We ensured that it is by extending it
Layout of an extension

- 3 small extensions
- 2 ½ weeks coding (no prior experience with the codebase)
- ~1000 lines of code
- In self-contained directory structure
Layout of an extension

- *ExtensionInfo* is sub-classed for each extension.
  - Calls a new scanner and an extended parser
  - Creates factories for creating Polyglot AST nodes and type objects
  - (Re)Orders the passes of the compiler
The Cast Pointcut

• Defines a new shadow join point encompassing each explicit or implicit cast, and a pointcut to match it

• Syntax:

```
cast ( TypePattern )
```

matches all casts to a type matching the `TypePattern`
The Cast Pointcut

• For example

    pointcut int_to_short(int x) :
    cast(short) && args(x);

• matches a cast from an int to a short and binds x to the original int
Check bounds with Cast Pointcut

```java
import uk.ac.ox.comlab.abc.eaj.lang.reflect.CastSignature;

aspect BoundsCheck
{
    before(int x) :
        cast(short) && args(x)
    {
        CastSignature s = (CastSignature)
            thisJoinPointStaticPart.getSignature();

        if (x > Short.MAX_VALUE || x < Short.MIN_VALUE) {
            System.out.println("Warning: information lost casting "+
                x + " to a " + s.getCastType().getName());
        }
    }
}
```
class LoseInformation
{
    public static void main(String[] args)
    {
        int x = 50000;
        short y;
        y = (short) x;
    }
}

$ java LoseInformation

Warning: information lost casting 50000 to a short
Implementing the Cast Pointcut

- **Frontend**
  - New polyglot AST node: \textit{PCCast}

- **Backend**
  - Cast pointcut class
  - Cast shadow join point class

- **Runtime**
  - Cast signature
Implementing the Cast Pointcut

- Create a polyglot AST node which stores the `TypePattern`

```java
Class PCCast_c extends Pointcut_c
    implements PCCast
{
    protected TypePatternExpr type_pattern;
    .
    .
    public abc.weaving.aspectinfo.Pointcut makeAIPointcut()
    {
        return new
        abc.eaj.weaving.aspectinfo.Cast
            (type_pattern.makeAITypePattern(), position());
    }
}
```
Implementing the Cast Pointcut

- The cast pointcut matches cast join points if they cast a type matching a `TypePattern`

```java
class Cast extends ShadowPointcut {
    private TypePattern type_pattern;
    
    protected Residue matchesAt(ShadowMatch sm) {
        if (! (sm instanceof CastShadowMatch)) return null;
        Type cast_to = ((CastShadowMatch) sm).getCastType();
        if (!getPattern().matchesType(cast_to)) return null;
        return AlwaysMatch.v;
    }
}
```
Implementing the Cast Pointcut

- Casts only occur on the right-hand-side of assignments in Jimple

```java
class CastShadowMatch extends StmtShadowMatch {
    private Type cast_to;
    
    public static CastShadowMatch(MethodPosition pos) {
        if (!(pos instanceof StmtMethodPosition)) return null;
        
        if (!(rhs instanceof CastExpr)) return null;
        Type cast_to = ((CastExpr) rhs).getCastType();
        return new CastShadowMatch(pos.getContainer(), stmt, cast_to);
    }
}
```
Implementing the Cast Pointcut

- *CastSignature*, in the runtime library, allows the retrieval of the type of a cast at runtime
- The information needed by the runtime is encoded by the compiler in the same way that ajc does
Future extensibility

- AspectJ
  - When making compiler extensions you often want to change a class in the compiler source.
  - If you do, this leads to maintenance problems.
  - If you don't, you may have to subclass whole class hierarchies.
  - A possible solution is to use Intertype declarations.
abc summary

- where we are and where we're going -
abc and abcTests.xml

abcTests.xml includes all of ajcTests.xml, plus new tests

running abc on abcTests.xml:
865 passed
19 failed (9 from ajcTests.xml)
103 skipped

reasons for failure:
9 abc bugs
6 polyglot bugs
1 javaToJimple/soot bug
3 queries for ajc

reasons for skip:
18 no incremental compilation
    for aspects & aspect-aware classes
34 options
9 not compiled by javac
2 package dir mismatch
34 “known limitation” of ajc
6 scanner

skipped options: incremental, usejavac, strict, X0codesize, extdirs
skip compile attribute: aspectpath
Initial performance experiments

Speedup factor of abc over ajc (JIT)

5.2.2 discrepancy due to architecture-specific JIT optimisations
Development plans

- eliminate bugs
- improve compilation speed
- Java 1.5 support
- *J tagger for performance measurement
- Dava support for decompiling AspectJ to Java
- visualisation in Eclipse
Future optimisations & analyses

• Further around optimisations:
  – smart cycle breaking, inliner
• Interprocedural analysis for eliminating cflow overheads
• Test for pure aspects
• Slicer for AspectJ
Future language extensions

- Semantic pointcuts:
  - predicted cflow
  - dataflow pointcuts
  - tracecuts

- Feature composition
  - CCC/Plainway ideas integrated with AspectJ