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 \(\rightarrow\) advice method
  - same return type
  - arguments matching the advice formals
    - plus arguments for thisJoinpoint etc.
- \texttt{proceed} \texttt{statement} \(\rightarrow\) 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, ...);
    }
}
```
[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

- \texttt{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
Transferring joinpoint context (3)

```java
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)

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
Skip Flag

- Example program

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

aspect 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();
        }
    }
    ...
}
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(
                ..., // skip flag
                true,
                ...
            );
        }
    }
    public void bar(int i) {}
}
Alternative bindings

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
Alternative bindings (2)

```java
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);
                    break label_0;
                }
            }
            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
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