Formal AOP: Opportunity Abounds

James Riely

http://www.depaul.edu/~jriely

DePaul CTI, Chicago, USA

Much of this talk reports on joint work with

Glen Bruns

Radha Jagadeesan

Alan Jeffrey

Thanks for Inviting Me

I will try to say something interesting.

- Waffle.
  - Limiting the power of AOP — Equational Reasoning
- Cheese and Ham.
  - Class-based AOP and Weaving (with types)
  - “Pure” AOP
- Waffle.
  - Increasing the power of AOP — Temporal Logics

Focus of attention: aspects as method/function call interceptors.

Opening Waffle

The “Right” Abstractions

More complex programs require more expressive abstractions (ie, better tools).

- FORTRAN/ALGOL: expressions/recursive functions
- Structured Programming: first order control structures
- Labelled Break Statements/Exceptions: finally eliminate goto
- Higher-Order Programming: programmable control structures
- Modules/OO Programming: encapsulation of data and control
- Patterns: popularize higher-order OO
- AO Programming: encapsulation of “concerns” (Flavors)

Concerns

So what are we concerned about?

- Primary functionality (in its many aspects)
- Synchronization
- Persistence/Distribution
- User Interfaces
- Caching
- Security
- ...

How do we code using OOP/FP?

OOP/FP Solutions

- Hooks (Publish/Subscribe, Visitors) — must be placed ahead
- Wrappers (Decorators) — can be circumvented

AOP to the Rescue

- Obliviousness — no need to plan ahead
- Quantification — no way to circumvent

Why Aren’t We All Programming in Prolog?

Programming with quantification is a pain.

Why Aren’t We All Programming in Assembly Language?

Programming without equational reasoning is a pain.

Why Aren’t We All Programming in the Pi Calculus?

Same question.

Abstractions of the language need to support the way we work.

AOP: The Declarative Imperative

Fillman and Friedman: *The cleverness of classical AOP is augmenting conventional sequentiality with quantification, rather than supplanting it wholesale.*

- How can we reasonably quantify over programs?
- How can we reason about programs over which we quantify?

Obliviousness is a two edged sword:

- Code providers should be oblivious to aspects — attach them where you like
- Code clients should be oblivious to aspects — assure that contracts will be validated

In both cases equational reasoning is essential.

Aspects Break Equational Reasoning: I

```
              class C { void foo() { } }
class D1 extends C { }
class D2 extends C { void foo() { super.foo(); } }

aspect Diff {
   void around(): execution(D.foo()) {
      System.out.println("aspect in action");
   }
}
```

D1.foo() $\neq$ D2.foo().

Aspects Break Equational Reasoning: II

```
class E1 {
  void f() { f(); }
  void g() { g(); }
}
class E2 {
  void f() { g(); }
  void g() { f(); }
}
aspect Diff {
   void around(): execution(E.f()) {
      System.out.println("aspect in action");
   }
}
```

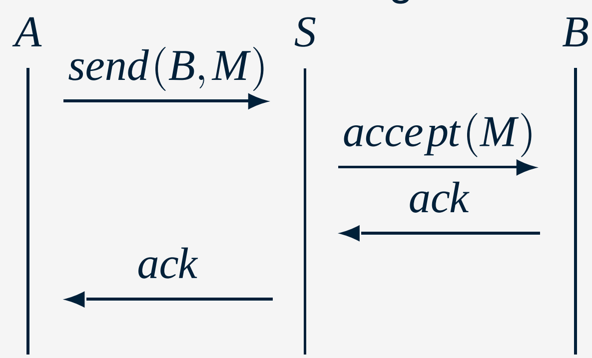
`E1.f()` $\neq$ **`E2.f()`**.

Also consider “jumping” and “vanishing” aspects.

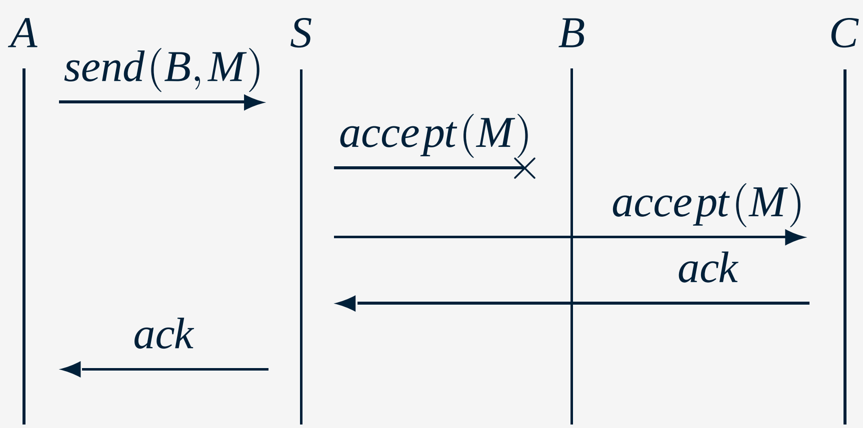
(example from Mitch Wand)

Aspects Interfere with Each Other

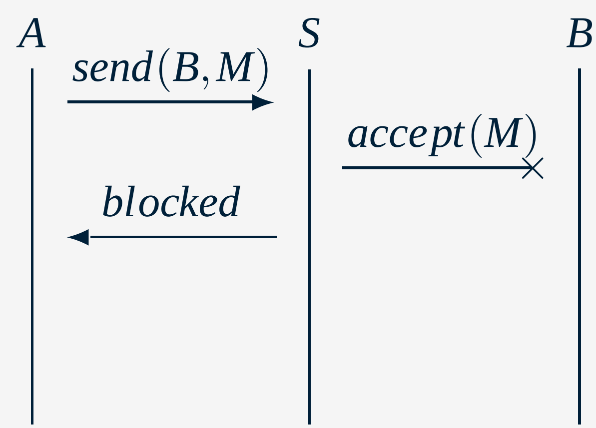
Alice calls Bob using a Server



Bob Forwards to Charlie



Bob blocks calls from Alice



WWDD?

Are aspects the new **`goto`**?

- **`goto`** problem “solved” by finding sufficiently expressive abstractions for control.
- Sanity of Hoare Logic mostly restored.
- Aspects will inevitably follow the same path. (Much work done in this direction, eg [Aldrich, thirty minutes ago].)
- [Wand ICFP 2003]: Need general support for domain-specific aspect languages. Need specification-level joint-point ontologies (AspectJ is implementation level.)
- Connections with behavioral types, behavioral subtyping.
- Contextual equivalence [Gordon’s applicative bisimulation] as useful tool. What are the observable events?

A Continuum of Approaches

- Meta-Object Protocols/Full-blown Introspection with Intercession
  - Compile-time
  - Load-time
  - Run-time
- Clearbox AOP (a lá AspectJ [Kiczales, et al])
- Blackbox AOP (a lá Composition Filters [Aksit, et al])
- Domain-Specific AOP
- Traditional OO/FP

What is the sweet spot?

AOP in the Wild Wild West

AOP is exploring its power.

Wither formal aspects of aspects?

- Local sheriff — calls it like it is
- School marm — drawing in the reigns
- Stranger without name — enabling new conquests
  - Hooker with heart of gold, if you prefer

Some Examples (Quickly)

Lopes Example: Bounded Buffer

DJ	JAVA

<pre>public class BoundedBuffer {
  private Object array[];
  private int putPtr = 0, takePtr = 0;
  private int usedSlots=0;
 	
  public BoundedBuffer(int capacity) {
    array = new Object[capacity];
  }
 	
  public void put(Object o) {
    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;
    usedSlots++;
  }
 	
  public Object take() {
    Object old = array[takePtr];
    array[takePtr] = null;
    takePtr = (takePtr + 1) % array.length;
    usedSlots--;
    return old;
  }
}
 	
coordinator BoundedBuffer {
 selfex put, take;
 mutex {put, take};
 cond full = false, empty = true;
 put: requires !full;
   on_exit {
     empty = false;
     if (usedSlots == array.length)
         full = true;
   }
 take: requires !empty;
     on_exit {
       full = false;
       if (usedSlots == 0) empty = true;
     }
}</pre>
	<pre>public class BoundedBuffer {
	  private Object[] array;
	  private int putPtr = 0, takePtr = 0;
	  private int usedSlots = 0;
	
	  public BoundedBuffer (int capacity) {
	    array = new Object[capacity];
	  }
	
	  public synchronized void put(Object o) {
	    while (usedSlots == array.length) {
	      try {
	        wait();
	      }
	      catch (InterruptedException e) {};
	    }
	    array[putPtr] = o;
	    putPtr = (putPtr + 1) % array.length;
	    }
	    if (usedSlots++ == 0)
	      notifyAll();
	  }
	
	  public synchronized Object take() {
	    while (usedSlots == 0) {
	      try {
	        wait();
	      }
	      catch (InterruptedException e) {};
	    }
	    Object old = array[takePtr];
	    array[takePtr] = null;
	    takePtr = (takePtr+1) % array.length;
	    }
	    if (usedSlots-- == array.length)
	      notifyAll();
	    return old;
	  }
	}</pre>

Lopes Example: Distributed Book Locator

DJ

```
portal BookLocator {
 void register (Book book, Location l);
 Location locate (String title)
 default:
   Book: copy{Book only title,author,isbn;}
}
portal Printer {
  void print(Book book) {
    book: copy { Book only title,ps; }
  }
}

class Book {
  protected String title, author;
  protected int isbn;
  protected OCRImage firstpage;
  protected Postscript ps;
  // All methods omitted
}
class BookLocator {
  // books[i] is in locations[i]
  private Book     books[];
  private Location locations[];
  // Other variables omitted
  public void register(Book b, Location l){
    // Verify and add book b to database
  }
  public Location locate (String title) {
    Location loc;
    // Locate book and get its location
    return loc;
  }
  // other methods omitted
}
class Printer {
  public void print(Book b) {
    // Print the book
  }
}

coordinator BookLocator {
  selfex register;
  mutex {register, locate};
}
```

JAVA

```
interface Locator extends Remote {
 void register(String title,
               String author, int isbn,
               Location l)
      throws RemoteException;
 Location locate(String title)
      throws RemoteException;
}
interface PrinterService extends Remote {
  void print(String title, Postscript ps)
       throws RemoteException;
}
class Book {
  protected String title, author;
  protected int isbn;
  protected OCRImage firstpage;
  protected Postscript ps;
  // All methods omitted
}
class BookLocator
        extends UnicastRemoteObject
        implements Locator {
  // books[i] is in locations[i]
  private Book     books[];
  private Location locations[];
  // Other variables omitted
  public void register (String title,
                        String author,
                        int isbn,
                        Location l)
         throws RemoteException {
    beforeWrite(); //for synchronization
    Book b=new Book (title, author, isbn);
    // Verify and add book b to database
    afterWrite(); //for synchronization
  }
  public Location locate (String title)
         throws RemoteException {
    Location loc;
    beforeRead(); //for synchronization
    // Locate book and get its location
    afterRead(); //for synchronization
    return loc;
  }
  // other methods omitted
}
class Printer extends UnicastRemoteObject
              implements PrinterService {
  public void print(String title,
                    Postscript ps)
         throws RemoteException {
    // Print the book
  }
}
```

Walker Example: Composable Security

```
fileNotNetwork =
{
  actions:  File.*, Network.*;
  policy:
    next →
      case ⋆ of
        File.* → run (filePolicy)
        Network.* → halt
      end
    done →  ()
}
networkNotFile =
{
  actions:  File.*, Network.*;
  policy:
    next →
      case ⋆ of
        File.* → halt
        Network.* → run (networkPolicy)
      end
    done →  ()
}
ChineseWall = fileNotNetwork ∨τ networkNotFile
```

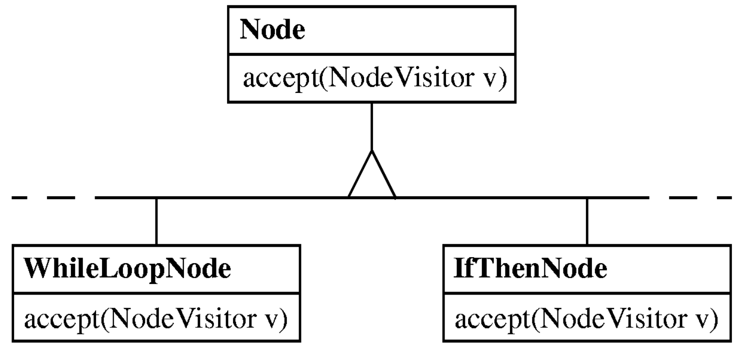
Aldrich Example: Dynamic Programming

```
val fib = fn x:int => 1
around call(fib) (x:int) =
    if (x > 2)
        then fib(x-1) + fib(x-2)
        else proceed x

(* advice to cache calls to fib *)
val inCache = fn ...
val lookupCache = fn ...
val updateCache = fn ...

pointcut cacheFunction = call(fib)
around cacheFunction(x:int) =
    if (inCache x)
        then lookupCache x
        else let v = proceed x
            in updateCache x v; v
```

Figure 2: The Fibonacci function written in `TinyAspect`, along with an aspect that caches calls to `fib`.

Clifton/Leavens Example: Visitors are Painful

```
public class WhileLoopNode extends Node {
  protected Node condition, body;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitWhileLoop(this);
  }
}
public class IfThenNode extends Node {
  protected Node condition, thenBranch;
  /* ... */
  public void accept(NodeVisitor v) {
    v.visitIfThen(this);
  }
}
```




```
public abstract class NodeVisitor {
  /* ... */
  public abstract void visitWhileLoop(WhileLoopNode n);
  public abstract void visitIfThen(IfThenNode n);
}
public class TypeCheckingVisitor extends NodeVisitor {
  /* ... */
  public void visitWhileLoop(WhileLoopNode n) { n.getCondition().accept(this); /* ... */ }
  public void visitIfThen(IfThenNode n) { /* ... */ }
}
```

Figure 1: Java code for some participants in the Visitor design pattern

```
// Methods for typechecking
public boolean Node.typeCheck()
  { /* ... */ }
public boolean WhileLoopNode.typeCheck()
  { /* ... */ }
public boolean IfThenNode.typeCheck()
  { /* ... */ }
```

Flatt/Krishnamurthi/Felleisen Example: Mixins as Wrappers

```
class LockedDoor^c extends Door^c {
  boolean canOpen(Person^c p) {
     if (!p.hasItem(theKey)) {
       System.out.println("You don't have the Key");
       return false;
     }
     System.out.println("Using key...");
     return super.canOpen(p);
  }
}
class ShortDoor^c extends Door^c {
  boolean canPass(Person^c p) {
     if (p.height() > 1) {
       System.out.println("You are too tall");
       return false;
     }
     System.out.println("Ducking into door...");
     return super.canPass(p);
  }
}

/* Cannot merge for LockedShortDoor^c */
```

```
interface Door^i {
  boolean canOpen(Person^c p);
  boolean canPass(Person^c p);
}
mixin Locked^m extends Door^i {
   boolean canOpen(Person^c p) {
      if (!p.hasItem(theKey)) {
        System.out.println("You don't have the Key");
        return false;
      }
      System.out.println("Using key...");
      return super.canOpen(p);
   }
}
mixin Short^m extends Door^i {
   boolean canPass(Person^c p) {
      if (p.height() > 1) {
        System.out.println("You are too tall");
        return false;
      }
      System.out.println("Ducking into door...");
      return super.canPass(p);
   }
}
class LockedDoor^c = Locked^m(Door^c);
class ShortDoor^c = Short^m(Door^c);
class LockedShortDoor^c = Locked^m(Short^m(Door^c));
```

Fig. 9. Some class definitions and their translation to composable mixins

Semantics

Understanding Pointcuts and Advice

Much work has been done.

- Connections with other things: Predicate Dispatching, Multimethods, MOPs, Reflection, Dynamically Scoped Functions, Subject Oriented Programming, *Coordination Languages?*, *Logic and constraint programming?*
- Semantics: Denotational, Big-step operational, Small-step operational, Haskell, Scheme, Common Lisp. Eg, [de Meuter], [Andrews], [Douence Motelet Sudholt], [Lämmel], [Wand Kiczales Dutchyn], [Masuhara Kiczales Dutchyn], [Walker Zdancewic Ligatti]
- Emphasis on understanding context-dependent pointcuts (**`cflow`**). Eg, [Wand Kiczales Dutchyn 2002].
- Our work: Emphasis on difference between pointcuts that fire before and after a call. Closest related work is [Lämmel 2002].

A Calculus of AO Programs (ECOOP 2003)

- Direct semantics of class-based and aspect-based languages.
- Small core of orthogonal primitives in ABL.
  - Only **`around`** advice — encode **`before`** and **`after`**
  - No method bodies — only advice bodies
  - Only call/execution pointcuts — and boolean connectives
- Concurrency and nested declarations are easy.
- Punted advice ordering: assume a global order on names.
- Specification of weaving and proof of correctness (in absence of dynamically arriving advice).

Specification of Weaving

No reductions are lost:

$$\text{complete}\quad \begin{array}{ccc} P_{\mathrm{ABL}} & \overset{weave}{\Longrightarrow} & P_{\mathrm{CBL}} \\ \Downarrow_{\triangledown} & & \\ P'_{\mathrm{ABL}} & & \end{array} \quad\text{as}\quad \begin{array}{ccc} P_{\mathrm{ABL}} & \overset{weave}{\Longrightarrow} & P_{\mathrm{CBL}} \\ \downarrow_{\triangledown} & & \downarrow \\ P'_{\mathrm{ABL}} & \overset{weave}{\Longrightarrow} & P'_{\mathrm{CBL}} \end{array}$$

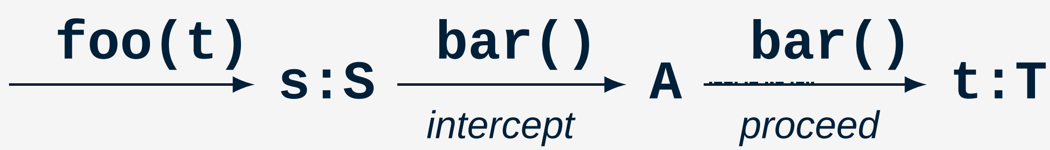
No reductions are gained:

$$\text{complete}\quad \begin{array}{ccc} P_{\mathrm{ABL}} & \overset{weave}{\Longrightarrow} & P_{\mathrm{CBL}} \\ & & \downarrow \\ & & P'_{\mathrm{CBL}} \end{array} \quad\text{as}\quad \begin{array}{ccc} P_{\mathrm{ABL}} & \overset{weave}{\Longrightarrow} & P_{\mathrm{CBL}} \\ \downarrow_{\triangledown} & & \downarrow \\ P'_{\mathrm{ABL}} & \overset{weave}{\Longrightarrow} & P'_{\mathrm{CBL}} \end{array}$$

($\rightarrow$ is OO reduction; $\rightarrow_{\triangledown}$ is AO reduction)

Example: s delegates to t

```
class S {
  void print() { out.print("I am a S"); }
  void foo(T t) { t.bar(); }
}
class T {
  void print() { out.print("I am a T"); }
  void bar() { }
}
advice A at call(T.bar()) {
  out.print("Aspect invoked");
  proceed();
}
```



A intercepts the message.

Call Advice

```
class S {
  void print() { out.print("I am a S"); }
  void foo(T t) { t.bar(); }
}
class T {
  void print() { out.print("I am a T"); }
  void bar() { }
}
protected S advice A at call(T.bar()) {
  this.print();
  target.print();
  proceed();
}
```

`s.foo(t)` prints “**`I am S; I am T`**”.

Call advice executed in the controlling context of the caller

Exec Advice

```
class S {
  void print() { out.print("I am a S"); }
  void foo(T t) { t.bar(); }
}
class T {
  void print() { out.print("I am a T"); }
  void bar() { }
}
protected T advice A at exec(T.bar()) {
  this.print();
  target.print();
  proceed();
}
```

`s.foo(t)` prints "**`I am T; I am T`**".

Exec advice executed in the controlling context of the callee

The Class Calculus: Some Reductions

- Field get

$$\text{object } o : c \{ \ldots f = v \ldots \} \quad \text{thread } \{ \text{let } x = o.f ; \vec{C} \} \;\rightarrow\; \text{object } o : c \{ \ldots f = v \ldots \} \quad \text{thread } \{ \text{let } x = v ; \vec{C} \}$$

- Field set

$$\text{object } o : c \{ \ldots f = u \ldots \} \quad \text{thread } \{ \text{set } o.f = v ; \vec{C} \} \;\rightarrow\; \text{object } o : c \{ \ldots f = v \ldots \} \quad \text{thread } \{ \vec{C} \}$$

- New declarations

$$\text{thread } \{ \text{new class } c <: d \{ \ldots \} ; \text{ object } o : c \{ \ldots \} ; \vec{C} \} \;\rightarrow\; \text{class } c <: d \{ \ldots \} \quad \text{object } o : c \{ \ldots \} \quad \text{thread } \{ \vec{C} \}$$

The Class Calculus: Method call

$\mathsf{class}\ \mathsf{d} \mathrel{\mathbf{<:}} \mathsf{Object}\ \{ \ldots \mathsf{m}\,\mathbf{(}\mathsf{x}\mathbf{)}\ \{ \vec{B} \} \ldots \}$

$\mathsf{class}\ \mathsf{c} \mathrel{\mathbf{<:}} \mathsf{d}\ \{ \ldots \}$

$\mathsf{object}\ \mathsf{o} \mathbf{:} \mathsf{c}\ \{ \ldots \}$

$\mathsf{thread}\ \{ \mathsf{o}\mathbf{.}\mathsf{m}\,\mathbf{(}\mathsf{v}\mathbf{)}\,\mathbf{;} \vec{C} \}$

$\longrightarrow$

$\mathsf{class}\ \mathsf{d} \mathrel{\mathbf{<:}} \mathsf{Object}\ \{ \ldots \mathsf{m}\,\mathbf{(}\mathsf{x}\mathbf{)}\ \{ \vec{B} \} \ldots \}$

$\mathsf{class}\ \mathsf{c} \mathrel{\mathbf{<:}} \mathsf{d}\ \{ \ldots \}$

$\mathsf{object}\ \mathsf{o} \mathbf{:} \mathsf{c}\ \{ \ldots \}$

$\mathsf{thread}\ \{ \vec{B}[{}^{\mathsf{o}}/_{\mathsf{this}}, {}^{\mathsf{v}}/_{\mathsf{x}}]\,\mathbf{;} \vec{C} \}$

The Aspect Calculus

- A pointcut ϕ is an element of the boolean algebra with atoms:
  - call $(c :: m)$
  - exec $(c :: m)$
- An advice declaration D binds message arguments $\vec{x}$ as well as this and target.
  - advice $a(\vec{x})$ at $\phi \ \{ \vec{C} \}$
- A class declaration D list the methods of the class (no code)
  - class c <: $d \ \{ m_1, m_2 \ldots \}$
- New commands C are:
  - let $x = o[\bar{a}\,;\bar{b}](\vec{v})$; process call advice $\bar{a}$ and exec advice $\bar{b}$.
  - let $x =$ proceed$(\vec{v})$; proceed to next advice

Supporting Call advice

- To implement call advice a lá AspectJ, record the static type of object references on method calls:

$$\text{let } x \texttt{=} o \texttt{:} c \texttt{.} m \texttt{(} \vec{v} \texttt{);}$$

- To bind this in call advice, record the controlling object of a thread:

$$\text{thread } p \ \{ S \}$$

- These changes are required to implement the dynamic semantics.

Aspect Reduction: Context

$\mathsf{advice}\ a_0(x) : \mathsf{call}(c :: m)\ \{\ \vec{C}_0\ \}$

$\mathsf{advice}\ a_3(x) : \mathsf{call}(d :: m)\ \{\ \vec{C}_3\ \}$

$\mathsf{advice}\ b_1(x) : \mathsf{exec}(c :: m)\ \{\ \vec{C}_1\ \}$

$\mathsf{advice}\ b_2(x) : \mathsf{exec}(d :: m)\ \{\ \vec{C}_2\ \}$

$\mathsf{object}\ o : d\ \{ \ldots \}$

$\mathsf{class}\ d <: c\ \{ \ldots \}$

$\mathsf{thread}\ p\{\ \mathsf{let}\ x = o : c.m(v);\ \}$

Actual type of o is d.

Declared type of o in thread is c.

Aspect Reduction: Fetching Advice

$$\text{advice } a_0\,(x) : \text{call}(c :: m)\ \{\vec{C}_0\}$$

$$\text{advice } a_3\,(x) : \text{call}(d :: m)\ \{\vec{C}_3\}$$

$$\text{advice } b_1\,(x) : \text{exec}(c :: m)\ \{\vec{C}_1\}$$

$$\text{advice } b_2\,(x) : \text{exec}(d :: m)\ \{\vec{C}_2\}$$

$$\text{object } o : d\ \{\ldots\}$$

$$\text{class } d <: c\ \{\ldots\}$$

$$\text{thread } p\{\text{let } x = o : c.m(v);\}$$

$$\twoheadrightarrow$$

$$\text{thread } p\{\text{let } x = o.[a_0 ; b_1, b_2](v);\}$$

Aspect Reduction: Call Advice

$$\text{advice } a_0\,(x) : \text{call}(c :: m)\ \{\vec{C}_0\}$$

$$\text{advice } a_3\,(x) : \text{call}(d :: m)\ \{\vec{C}_3\}$$

$$\text{advice } b_1\,(x) : \text{exec}(c :: m)\ \{\vec{C}_1\}$$

$$\text{advice } b_2\,(x) : \text{exec}(d :: m)\ \{\vec{C}_2\}$$

$$\text{object } o : d\ \{\ldots\}$$

$$\text{class } d <: c\ \{\ldots\}$$

$$\text{thread } p\{\text{let } x = o.[a_0 ; b_1, b_2](v);\}$$

$$\rightarrow\!\!\!\rhd$$

$$\text{thread } p\{\text{let } x = p\{\vec{C}_0[{}^{v}\!/\!_{x}, {}^{p}\!/\!_{\text{this}}, {}^{o}\!/\!_{\text{target}}, {}^{o.[\emptyset ; b_1, b_2]}\!/\!_{\text{proceed}}]\};\}$$

Controlling context is p.

Aspect Reduction: Exec Advice

$$\text{advice } a_0\,(x) : \text{call}(c :: m)\ \{\ \vec{C}_0\ \}$$

$$\text{advice } a_3\,(x) : \text{call}(d :: m)\ \{\ \vec{C}_3\ \}$$

$$\text{advice } b_1\,(x) : \text{exec}(c :: m)\ \{\ \vec{C}_1\ \}$$

$$\text{advice } b_2\,(x) : \text{exec}(d :: m)\ \{\ \vec{C}_2\ \}$$

$$\text{object } o : d\ \{\ \ldots\ \}$$

$$\text{class } d <: c\ \{\ \ldots\ \}$$

$$\text{thread } p\{\ \text{let } x = o.[\emptyset ; b_1, b_2](v);\ \}$$

$$\rightarrow$$

$$\text{thread } p\{\ \text{let } x = o\{\ \vec{C}_1[{}^{v}/_{x}, {}^{o}/_{this}, {}^{o}/_{target}, {}^{o.[\emptyset ; b_2]}/_{proceed}]\ \};\ \}$$

Controlling context is o.

Encoding the CBL into the ABL

- Given a class:

$$\text{class } c \mathbf{<:} \text{ Object } \{ \ldots m\mathbf{(}\vec{x}\mathbf{)} \ \{ \vec{C}_0 \} \ldots \}$$

$$\text{class } d \mathbf{<:} \ c \ \{ \ldots m\mathbf{(}\vec{x}\mathbf{)} \ \{ \vec{C}_1 \} \ldots \}$$

- Create exec advice for each body:

$$\text{advice cbl\_}c\text{\_}m\mathbf{(}\vec{x}\mathbf{)} \mathbf{:} \text{exec}\mathbf{(}d \mathbf{::} m\mathbf{)} \ \{ \vec{C}_0 [{}^{\text{proceed}}/_{\text{super}.m}] \}$$

$$\text{advice cbl\_}d\text{\_}m\mathbf{(}\vec{x}\mathbf{)} \mathbf{:} \text{exec}\mathbf{(}d \mathbf{::} m\mathbf{)} \ \{ \vec{C}_1 [{}^{\text{proceed}}/_{\text{super}.m}] \}$$

- Ensure that cbl_d_m has higher priority than cbl_c_m.
- More robust encoding of super uses static dispatch directly.

Weaving

- Programs that dynamically load advice affecting existing classes cannot be woven statically.
- For static advice, weaving is something like macro expansion:

$$\text{class } c \mathbin{\texttt{<:}} d \;\{\, m\,[\,\emptyset\,;\, b_1, b_2\,]\,\}$$

$$\text{advice } b_1\,(\vec{x}) : \text{exec}(d :: m)\;\{\, \vec{C}_1 \,\}$$

$$\text{advice } b_2\,(\vec{x}) : \text{exec}(d :: m)\;\{\, \vec{C}_2 \,\}$$

is woven recursively as

$$\text{class } c \mathbin{\texttt{<:}} \ldots \;\{\, m\,(\vec{x})\;\{\, \vec{C}_1 [{}^{\text{this}}/_{\text{target}},\, {}^{\text{this.}[\emptyset\,;\,b_2]}/_{\text{proceed}}] \,\}\,\}$$

$$\text{advice } b_2\,(\vec{x}) : \text{exec}(d :: m)\;\{\, \vec{C}_2 \,\}$$

- The terminating version of this idea is now standard.

Weaving: Subtleties

- Extra parameter on call advice (for target object)
- Knowledge of controlling object required for call advice
- Must annotate advised method calls with method name (required for switch from call to exec advice)
- Introduce skip step to match advice lookups (required so that reductions match one-to-one)
- Theorem works modulo an equivalence on names (weaving must use actual method name, but aspect code uses name based on advice list)

complete
$$\begin{array}{ccc} P_{\mathrm{ABL}} & \stackrel{weave}{\Longrightarrow} & P_{\mathrm{CBL}} \\ \downarrow & & \\ P'_{\mathrm{ABL}} & & \end{array}$$
as
$$\begin{array}{cccc} P_{\mathrm{ABL}} & \stackrel{weave}{\Longrightarrow} & & P_{\mathrm{CBL}} \\ \downarrow & & & \downarrow \\ P'_{\mathrm{ABL}} & \stackrel{weave}{\Longrightarrow} & \simeq & P'_{\mathrm{CBL}} \end{array}$$

The Full Untyped AOL

$a,..,z$	*Name*	C,B ::=	*Command*
P,Q ::= $(\overline{D} \vdash \overline{H})$	*Program*	new $\overline{D}\,\overline{H}$**;**	New Declaration
D,E ::=	*Declaration*	return v**;**	Return
class c **<:** d $\{\overline{M}\}$	Class	let x**=**v**;**	Value
advice a**(**$\vec{x}$**)** **:** ϕ $\{\vec{C}\}$	Advice	let x**=**o**.**f**;**	Get Field
M ::= m**[**$\overline{a}$; $\overline{b}$**]**	*Method*	set o**.**f**=**v**;**	Set Field
H,G ::=	*Heap Element*	let x**=**o**.**c**::**m**(**$\vec{v}$**);**	Static Message
object o**:**c $\{\overline{F}\}$	Object	let x**=**o**:**c**.**m**(**$\vec{v}$**);**	Dynamic Message
thread $o\{S\}$	Thread	let x**=**o**.**m**[**$\overline{a}$; $\overline{b}$**](**$\vec{v}$**);**	Advised Message
F ::= f**=**v	*Field*	let x**=**proceed**(**$\vec{v}$**);**	Proceed
S,T ::=	*Call Stack*	ϕ,ψ ::=	*Pointcut*
$\vec{C}$	Current Frame	false	False
let x**=**$o\{S\}$**;**$\vec{C}$	Pushed Frame	$\neg\phi$	Negation
		$\phi \vee \psi$	Disjunction
		call**(**c**::**m**)**	Call
		exec**(**c**::**m**)**	Execution

Types (Unpublished)

Typing is Problematic

A symptom: the following code compiles in AspectJ1.1.

```
class D {
  public String m() { return "D"; }
}
aspect A {
  Object around(): call(* D.m()) {
    return new Integer(1);
  }
}
```

This looks like a bug.

Real issues: modular typechecking, variance, genericity.

We address only the first issue.

$$\text{if } \vdash P \text{ and } \vdash Q \text{ then } \vdash P \mid Q$$

A Difference with AspectJ

- The set of call advice does not depend upon the type of the caller.
- To avoid locking entire heap on every method call, the declaration set is *closed* to precompute advice lists:

$$\text{class } c \mathbin{\texttt{<:}} \ldots \{\, m\,\texttt{[}\,\overline{a}\,\texttt{;}\,\overline{b}\,\texttt{]}, \ldots \}$$

- To allow modular typechecking and the use of this in call advice, must constrain the type of the caller.
- Method declarations have the form:

$$\text{class } c \mathbin{\texttt{<:}} \ldots \{\, \text{protected } s \text{ method } m\texttt{(}\vec{t}\texttt{)}\texttt{:}r\ \texttt{[}\,\overline{a}\,\texttt{;}\,\overline{b}\,\texttt{]} \ldots \}$$

- **`protected`** is "protected c"; **`public`** is "protected Object".

Another Difference

- In AspectJ, each advice list terminates in a call to a plain class, which cannot proceed.
- To capture this, we must distinguish two types of advice:

$\rho ::=$	*Placement*
around	Around
replace	Replace
$D, E ::= \ldots$	*Declaration*
ρ advice $a(\vec{x}:\vec{t}):r$ at $\phi\ \{\vec{C}\}$	Advice

Results for the Typed Calculus

The development is fairly standard

- Weaving still correct
- Weaving preserves types
- Reduction preserves types
- **`around`** advice no longer enough (**`before`** and **`after`** not encodable)

Lays the groundwork for

- Covariant return / Contravariant arguments
- Genericity
- Row polymorphism

The Full Typed AOL

$a,..,z$	*Name (& Type)*
X,Y,Z ::= n **:** t	*Typed Name*
P,Q ::= $(\overline{D} \vdash \overline{H})$	*Program*
ρ ::=	*Placement*
around	Around
replace	Replace
D,E ::=	*Declaration*
class c **<:** d { $\overline{F}$ $\overline{M}$ }	Class
ρ advice a**(**$\vec{X}$**):**r at ϕ { $\vec{C}$ }	Advice
M ::= protected s method m**(**$\vec{t}$**):**r **[**$\overline{a}$ **;** $\overline{b}$**]**	*Method*
F ::= protected s field f **:** t**;**	*Field Type*
V ::= f **=** v**;**	*Field Value*
H,G ::=	*Heap Element*
object o **:** c { $\overline{V}$ }	Object
thread o { S }	Thread
S,T ::=	*Call Stack*
$\vec{C}$	Current Frame
let X **=** o{ S }**;**$\vec{C}$	Pushed Frame

C,B ::=	*Command*
new $\overline{D}$ $\overline{H}$**;**	New
return v**;**	Return
let X **=** v**;**	Value
let X **=** o**.**f**;**	Get Field
set o**.**f **=** v**;**	Set Field
let X **=** o**.**c**::**m**(**$\vec{v}$**);**	Static Message
let X **=** o**:**c**.**m**(**$\vec{v}$**);**	Dynamic Msg
let X **=** o**:**c**.**m**[**$\overline{a}$**;** $\overline{b}$**](**$\vec{v}$**);**	Advised Msg
let X **=** proceed**(**$\vec{v}$**);**	Proceed
ϕ,ψ ::=	*Pointcut*
call**(**c**::**m**)**	Call
exec**(**c**::**m**)**	Execution
$\neg$call**(**c**::**m**)**	Not Call
$\neg$exec**(**c**::**m**)**	Not Execution
true	True
false	False
$\phi \wedge \psi$	Conjunction
$\phi \vee \psi$	Disjunction

μABC

$P, Q, R \ ::=$	*Program*
$\mathsf{let}\, x = p \to q : \vec{m}; P$	Message
$\mathsf{return}\, v$	Return
$\mathsf{role}\, p < q; P$	New Role
$\mathsf{advice}\ a[\phi] = \sigma x\,.\,\tau y\,.\,\pi b\,.\,Q; P$	New Advice

A Minimal Aspect-Based Calculus

Design Choices

Goals

- Really really small.
- Straightforward compositional translation of class-based language.

Decisions

- Start with Abadi and Cardelli's object calculus (σ).
- Add object hierarchy (each object beneath its creator).
- Remove everything else. Call objects *roles*.
- Remove asymmetry of OO. Message send has the form:

$$p \rightarrow q : \vec{m}$$

send messages $\vec{m}$ from p to q

Refactored Syntax

$f,\ldots,\ell,p,\ldots,z$	*Label or Role*
$a,\ldots,e$	*Advice name*
$m,n \ ::= \ell \mid a$	*Message*
$P,Q \ ::= \vec{B};\mathsf{return}\,v$	*Program*
$B,C \ ::= \mathsf{let}\,x = p \rightarrow q : \vec{m} \mid D$	*Command*
$D,E \ ::=$	*Declaration*
$\quad \mathsf{role}\,p < q$	Role
$\quad \mathsf{advice}\ a[\phi] = \sigma x\,.\,\tau y\,.\,\pi b\,.\,Q$	Advice

Advice names are not first class.

Pointcuts

- Syntax

$\phi, \psi ::=$	*Pointcut*
$p \to q : \ell$	Call
$\neg p \to q : \ell$	Not Call
$\phi \wedge \psi \mid \mathsf{true}$	Conjunction
$\phi \vee \psi \mid \mathsf{false}$	Disjunction
$\forall x \leq p \,.\, \phi$	Universal
$\exists x \leq p \,.\, \phi$	Existential

- Semantics

$$\vec{D} \vdash p \leq q$$

$$\vec{D} \vdash p \to q : \ell \ \mathsf{sat}\ \phi$$

Dynamic Semantics

$$\vec{D};\,\mathsf{let}\,z = p \to q : \vec{m}, \ell; P \;\rightarrowtriangle\; \vec{D};\,\mathsf{let}\,z = p \to q : \vec{m}, \vec{a}; P$$

$$\text{where } \langle \vec{a} \rangle = \langle\, a \mid \vec{D} \ni \mathsf{advice}\, a[\phi] \cdots \text{ and } \vec{D} \vdash p \to q : \ell \;\mathsf{sat}\; \phi \,\rangle$$

$$\vec{D};\,\mathsf{let}\,z = p \to q : \vec{m}, a; P \;\rightarrowtriangle\; \vec{D}; \vec{B}[{}^{p}\!/\!_{x}, {}^{q}\!/\!_{y}, {}^{\vec{m}}\!/\!_{b}]; P[{}^{v}\!/\!_{z}]$$

$$\text{where } \vec{D} \ni \mathsf{advice}\; a[\cdots] = \sigma x \,.\, \tau y \,.\, \pi b \,.\, \vec{B};\,\mathsf{return}\, v$$

Pick the rightmost message (for consistency with declaration order).

Renaming required in second rule — $\mathsf{dom}(\vec{B})$ and $\mathsf{fn}(P)$ disjoint.

Garbage collection $P \overset{\text{gc}}{\rightarrowtriangle} P'$ removes unused roles, advice, messages.

Sugar

Sugar on programs:

$$x \triangleq \text{return}\, x$$

$$p \rightarrow q : \vec{m} \triangleq \text{let}\, x = p \rightarrow q : \vec{m}; \text{return}\, x$$

$$\text{role}\, p \triangleq \text{role}\, p < \text{top}$$

Sugar on pointcuts:

$$p . \ell \triangleq \exists x \leq \text{top} . \exists y \leq p . x \rightarrow y : \ell$$

"$p . \ell$" fires when p or one of its subroles receives message ℓ.

Call-by-value Lambda Calculus

$$
\begin{array}{rcl}
\vec{D} & = & \mathsf{role}\, f;\\
& & \mathsf{advice}\; a[f\,.\,\mathsf{call}] = \tau y\,.\,\mathsf{let}\, x = y \rightarrow y : \mathsf{arg}; P;\\
& & \mathsf{role}\; g < f;\\
& & \mathsf{advice}\; b[g\,.\,\mathsf{arg}] = Q;
\end{array}
$$

$$
\begin{array}{rcl}
(\lambda x\,.\,P)\,Q & \twoheadrightarrow & \vec{D}; g \rightarrow g : \mathsf{call}\\
& \twoheadrightarrow & \vec{D}; g \rightarrow g : a\\
& \twoheadrightarrow & \vec{D}; \mathsf{let}\, x = g \rightarrow g : \mathsf{arg}; P\\
& \twoheadrightarrow & \vec{D}; \mathsf{let}\, x = g \rightarrow g : b; P\\
& \twoheadrightarrow & \vec{D}; \mathsf{let}\, x = Q; P\\
& \overset{\mathsf{gc}}{\twoheadrightarrow} & \mathsf{let}\, x = Q; P
\end{array}
$$

Cf. [Milner *Functions as Processes*]

Conditional

$$\begin{aligned}\text{if } p \le q \text{ then } R_1 \text{ else } R_2 \triangleq\ & \text{role } r;\\ & \text{advice } [\exists x \le \text{top} . x \to r : \text{if}] = R_2;\\ & \text{advice } [\exists x \le q . x \to r : \text{if}] = R_1;\\ & p \to r : \text{if}\end{aligned}$$

R_1 does not use its proceed variable. If R_1 fires, R_2 cannot fire.

$$\vec{D}; \text{if } p \le q \text{ then } R_1 \text{ else } R_2 \rightarrow^* \xrightarrow{\text{gc}} \begin{cases} R_1 & \text{if } \vec{D} \vdash p \le q \\ R_2 & \text{otherwise} \end{cases}$$

Lambda Calculus with Advice

We encode primitives from core MinAML [Walker Zdancewic Ligatti 2003]. See also [Tucker Krishnamurthi 2003].

- new $p; P$ creates a new name p which acts as a hook.
- $\{p \,.\, z \to Q\} \gg P$ attaches *after* advice $\lambda z \,.\, Q$ to hook p.
- $\{p \,.\, z \to Q\} \ll P$ attaches *before* advice $\lambda z \,.\, Q$ to hook p.
- $p\langle P \rangle$ evaluates P then runs advice hooked on p.

Not a full-blown translation. Eg, advice is first class in MinAML.

Core MinAML Reduction

$$P \triangleq \text{new}\ p; \{p.x_1 \to x_1+1\} \ll \{p.x_2 \to x_2*2\} \gg p\langle 3\rangle$$

$$\vec{D} \triangleq \text{role}\ p;$$
$$\text{advice}\ a[p.\text{hook}] = \lambda x_0 . x_0;$$
$$\text{advice}\ b[p.\text{hook}] = \tau z.\pi d.\lambda x_1 . \text{let}\ y_1 = x_1+1; (z \to z:d)(y_1);$$
$$\text{advice}\ c[p.\text{hook}] = \tau z.\pi d.\lambda y_2 . \text{let}\ x_2 = (z \to z:d)(y_2); x_2*2;$$

$$P = \vec{D}; (p \to p:\text{hook})\ 3$$
$$\twoheadrightarrow \vec{D}; (p \to p:a,b,c)\ 3$$
$$\twoheadrightarrow^* \xrightarrow{\text{gc}} \vec{D}; \text{let}\ x_2 = (p \to p:a,b)(3); x_2*2$$
$$\twoheadrightarrow^* \xrightarrow{\text{gc}} \vec{D}; \text{let}\ x_2 = (\text{let}\ y_1 = 3+1; (p \to p:a)(y_1)); x_2*2$$
$$\twoheadrightarrow^* \xrightarrow{\text{gc}} \vec{D}; \text{let}\ x_2 = (p \to p:a)(4); x_2*2$$
$$\twoheadrightarrow^* \xrightarrow{\text{gc}} \vec{D}; \text{let}\ x_2 = 4; x_2*2$$
$$\twoheadrightarrow^* \xrightarrow{\text{gc}} 8$$

Translating the CBL

$$[\![\text{advice } a[\phi](\vec{x})\ \{Q\}]\!] = \text{advice } a[[\![\phi]\!]] = \sigma\text{this} . \tau\text{target} . \pi\text{proceed} . \lambda\vec{x} . \text{this}[\![Q]\!]$$

$$[\![\text{class } t <: u\ \{\vec{M}\}]\!] = \text{role } t < u;\ \ t[\![\vec{M}]\!]$$

$$t[\![\text{method } \ell(\vec{x})\ \{Q\}]\!] = \text{advice } [t . \ell] = \tau\text{this} . \pi\text{super} . \lambda\vec{x} . \text{this}[\![Q]\!]$$

$$[\![\text{object } p : t\ \{\vec{F}\}]\!] = \text{role } p < t;\ \ p[\![\vec{F}]\!]$$

$$p[\![\text{field } f = v]\!] = \text{advice } a[\text{false}] = \text{return } v;$$

$$\text{advice } [p . f] = \sigma x . \tau y . \pi b . x \to y : a, b$$

$$p[\![\text{let } x = q.\ell(\vec{v}); P]\!] = \text{let } x = (p \to q : \ell)\,\vec{v}; p[\![P]\!]$$

Advice on fields; No call/exec distinction; No global advice order.

One step in CBL = Several steps in μABC (including garbage collection).

Insight from μABC

- Advice + Names + Name Substitution = Enough!
- Not much more complicated than λ, π or σ.
- Paper includes spaghetti CPS translation of μABC into π.
- Essence of class-based AOP: role hierarchy + advice binding source, target, and proceed.
- Are pure aspects efficiently implementable?

Closing Waffle

Motivating Example: Resource Access Control

- Access Matrix Model [Lampson 1974].

$$\text{Policy} : \text{Subject} \times \text{Object} \mapsto 2^{\text{Rights}}$$

- Stack Inspection [Wallach et al 1997].

$$\text{Stack} : \overrightarrow{\text{Subject}}$$

$$\text{Policy} : \text{Stack} \times \text{Object} \mapsto 2^{\text{Rights}}$$

- History-Based Access Control [Abadi Fournet 2003].

$$\text{Event} : \text{Subject} \times \text{Object} \times \text{Value} \times \{\mathsf{call}, \mathsf{return}\}$$

$$\text{History} : \overrightarrow{\text{Event}}$$

$$\text{Policy} : \text{History} \times \text{Object} \mapsto 2^{\text{Rights}}$$

Abadi/Fournet Example: Bad Plugin

```
// Trusted : static permissions contain all permissions.
public class NaiveProgram {
  public static void main() {
     String s = BadPlugIn.tempFile();
     new File(s).delete();
  }
}
// Mostly untrusted : static permissions don’t
// contain any FilePermission.
class BadPlugin {
  public static String tempFile() {
   return "..\\password";
  }
}
```

Aspects for Resource Access Control

- Access Matrix Model: **`call`**
- Stack Inspection: **`call`** + **`cflow`**
- History-Based: ?

A More General Notion of Past

- Connection between **`cflow`** and past-time eventuality operator $\diamond$ has been noted by many.
- **`cflow`**'s limitations are accepted on grounds of implementability.

How can we implement a more general notion of past?

- Required in Firewalls and Intrusion Detection Systems.
- An elegant solution: Security Automata [Schneider 2000].
- Idea: automaton maintains an *abstraction* of the history.

Sketching a Logic of Temporal Pointcuts

A logic based on regular expressions and process algebraic operators:

ε	empty.
$\phi;\psi$	sequential composition of two traces.
$\phi*$	closure of sequential composition — $\varepsilon \vee (\phi;\phi*)$.
$\phi \parallel \psi$	parallel composition of two traces.
$\phi!$	closure of parallel composition — $\varepsilon \vee (\phi \parallel \phi!)$.

Some encodings:

$$\mathsf{balanced} = (\mathsf{call};\mathsf{return})!$$

$$\mathsf{semi{\cdot}balanced} = (\mathsf{balanced};\mathsf{call}*)*$$

$$\mathsf{cflow}\langle\phi\rangle = (\phi \wedge \mathsf{call}*) \parallel \mathsf{balanced}$$

Challenges for Temporal Pointcuts

- Whose past? thread? caller object? callee object? stack?
- How does one handle partially completed methods and advice? At what point, exactly, does a call begin or end?
- What logics are implementable?
- Compile-time weaving no longer an option.
- Dynamically loaded aspects attractive – requires rebuilding the automaton (a new kind of weaving).
- What if new aspects require information that has not been saved?

Putting the Waffles Together

- Logics should be powerful enough to capture join points that are not recorded in the stack.
- Join points are themselves resources, whose access must be managed.
- Interference between aspect policies an important issue.
- Work on Feature Interaction is relevant.

Thank You!