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

```java
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");
    }
}
```

\[ \text{E1.f() \neq E2.f()}. \]

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
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

<table>
<thead>
<tr>
<th>DJ</th>
<th>JAVA</th>
</tr>
</thead>
</table>
| ```java
def 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;
    }
}
```
| ```java
def 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) {
        try {
            while (usedSlots == array.length) {
                wait();
            }
            array[putPtr] = o;
            putPtr = (putPtr + 1) % array.length;
            if (usedSlots++ == 0)
                notifyAll();
        }

        public synchronized Object take() {
            try {
                while (usedSlots == 0) {
                    wait();
                }
                Object old = array[takePtr];
                array[takePtr] = null;
                takePtr = (takePtr+1) % array.length;
                if (usedSlots-- == array.length)
                    notifyAll();
                return old;
            }
```
Lopes Example: Distributed Book Locator

### DJ

```java
portal

BookLocator {
  void register (Book book, Location l);
  Location locate (String title)
}

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[] is in locations[]
  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

```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[] is in locations[]
    private Book books[];
    private Location locations[];
    // Other variables omitted
    public void register (String title, String author, int isbn, Location l) {
      throws RemoteException;
    }
    public Location locate (String title) {
      Location loc;
      // Print the book
    }
  }
}

class Printer {
  extends UnicastRemoteObject
  implements PrinterService {
    public void print (String title, Postscript ps) {
      throws RemoteException;
    }
  }
```

FOAL '04 – p.17/6
Walker Example: Composable Security

```plaintext
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

```ocaml
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.
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()
    { /* ... */ }
class LockedDoor\textsuperscript{c} extends Door\textsuperscript{c} {
    boolean canOpen(Person\textsuperscript{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\textsuperscript{c} extends Door\textsuperscript{c} {
    boolean canPass(Person\textsuperscript{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);
    }
}

interface Door\textsuperscript{i} {
    boolean canOpen(Person\textsuperscript{i} p);
    boolean canPass(Person\textsuperscript{i} p);
}

class Locked\textsuperscript{m} extends Door\textsuperscript{i} {
    boolean canOpen(Person\textsuperscript{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 Short\textsuperscript{m} extends Door\textsuperscript{i} {
    boolean canPass(Person\textsuperscript{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 Locked\textsuperscript{c} = Locked\textsuperscript{m}(Door\textsuperscript{c});
class Short\textsuperscript{c} = Short\textsuperscript{m}(Door\textsuperscript{c});
class LockedShort\textsuperscript{c} = Locked\textsuperscript{m}(Short\textsuperscript{m}(Door\textsuperscript{c}));

/* Cannot merge for LockedShort\textsuperscript{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:

\[
\begin{align*}
& P_{ABL} \xrightarrow{\text{weave}} P_{CBL} \\
\text{complete} \quad & \quad \downarrow \\
& P'_{ABL} \\
\end{align*}
\]

\[
\begin{align*}
& P_{ABL} \xrightarrow{\text{weave}} P_{CBL} \\
\text{as} \quad & \quad \downarrow \\
& P'_{CBL} \\
\end{align*}
\]

No reductions are gained:

\[
\begin{align*}
& P_{ABL} \xrightarrow{\text{weave}} P_{CBL} \\
\text{complete} \quad & \quad \downarrow \\
& P'_{ABL} \\
\end{align*}
\]

\[
\begin{align*}
& P_{ABL} \xrightarrow{\text{weave}} P_{CBL} \\
\text{as} \quad & \quad \downarrow \\
& P'_{CBL} \\
\end{align*}
\]

(← is OO reduction; → 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();
}

\[ \text{foo}(t) \rightarrow s:S \quad \text{bar()} \rightarrow A \quad \text{bar()} \rightarrow t:T \]

\( A \) intercepts the message.
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
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();
}

\texttt{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
  
  \[
  \text{object } o : c \{ \ldots f = v \ldots \} \quad \Rightarrow \quad \text{object } o : c \{ \ldots f = v \ldots \}
  \]
  
  \[
  \text{thread } \{ \text{let } x = o.f; \vec{C} \} \quad \Rightarrow \quad \text{thread } \{ \text{let } x = v; \vec{C} \}
  \]

- Field set
  
  \[
  \text{object } o : c \{ \ldots f = u \ldots \} \quad \Rightarrow \quad \text{object } o : c \{ \ldots f = v \ldots \}
  \]
  
  \[
  \text{thread } \{ \text{set} o.f = v; \vec{C} \} \quad \Rightarrow \quad \text{thread } \{ \vec{C} \}
  \]

- New declarations
  
  \[
  \text{thread } \{ \text{new} \text{ class } c <: \text{d} \{ \ldots \}; \}
  \]
  
  \[
  \text{object } o : c \{ \ldots \}; \vec{C} \}
  \quad \Rightarrow \quad \text{class } c <: \text{d} \{ \ldots \}
  \]
  
  \[
  \text{object } o : c \{ \ldots \}; \vec{C} \}
  \quad \Rightarrow \quad \text{object } o : c \{ \ldots \}
  \]
  
  \[
  \text{thread } \{ \vec{C} \}
  \]
class d <: Object { ... m (x) \{ \vec{B} \} ... }
class c <: d { ... }
object o : c { ... }
thread \{ o . m (v) ; \vec{C} \}

\rightarrow

class d <: Object { ... m (x) \{ \vec{B} \} ... }
class c <: d { ... }
object o : c { ... }
thread \{ \vec{B}[^this, \forall x] ; \vec{C} \}
The Aspect Calculus

- A pointcut $\phi$ is an element of the boolean algebra with atoms:
  - call $(c :: m)$
  - exec $(c :: m)$

- An advice declaration $D$ binds message arguments $\bar{x}$ as well as this and target.
  - advice $a(\bar{x})$ at $\phi \{ \bar{C} \}$

- A class declaration $D$ list the methods of the class (no code)
  - class $c <: d \{ m_1, m_2, ... \}$

- New commands $C$ are:
  - let $x = o[\bar{a} ; \bar{b}] (\bar{v})$; process call advice $\bar{a}$ and exec advice $\bar{b}$.
  - let $x = \text{proceed} (\bar{v})$; proceed to next advice
Supporting Call advice

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

  \[
  \text{let } x = o : c \cdot m (\vec{v}) ;
  \]

- 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

advice a₀ (x) : call (c :: m) \{ \vec{C}_0 \}
advice a₃ (x) : call (d :: m) \{ \vec{C}_3 \}
advice b₁ (x) : exec (c :: m) \{ \vec{C}_1 \}
advice b₂ (x) : exec (d :: m) \{ \vec{C}_2 \}

object o : d \{ ... \}
class d <: c \{ ... \}

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

Actual type of o is d.
Declared type of o in thread is c.
aspect reduction: fetching advice

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

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

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

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

object \( o : d \) \{ ... \}

class \( d <: c \) \{ ... \}

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

\( \rightarrow \)

thread \( p \{ \text{let} \ x = o \cdot [a_0 ; b_1, b_2] \ (v) ; \} \)
Aspect Reduction: Call Advice

```
advice a_0 (x) : call (c :: m) \{ \vec{C}_0 \}
advice a_3 (x) : call (d :: m) \{ \vec{C}_3 \}
advice b_1 (x) : exec (c :: m) \{ \vec{C}_1 \}
advice b_2 (x) : exec (d :: m) \{ \vec{C}_2 \}
object o : d \{ ... \}
class d <: c \{ ... \}
```

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

```
\rightarrow
```

```
thread p\{ let x = p\{ \vec{C}_0[y/x, p/this, o/target, o. [\emptyset ; b_1,b_2]/proceed] \} ; \}
```

Controlling context is \( p \).
Aspect Reduction: Exec Advice

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

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

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

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

object \( o : d \) \{ ... \}

class \( d <: c \) \{ ... \}

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

\(\Rightarrow\)

thread \( p \{ \) let \( x = o \{ \vec{C}_1 [\backslash x, o /\text{this, } o /\text{target, } o \cdot [\emptyset ; b_2] /\text{proceed}] \} ; \) \}

Controlling context is \( o \).
Encoding the CBL into the ABL

Given a class:

```java
class c <: Object {
  ... m (\vec{x}) { \vec{C}_0 } ... 
}
```

```java
class d <: c {
  ... m (\vec{x}) { \vec{C}_1 } ... 
}
```

Create exec advice for each body:

```java
advice cbl_c_m (\vec{x}) : exec (d :: m) { \vec{C}_0[^{proceed/super.m}] }
advice cbl_d_m (\vec{x}) : exec (d :: m) { \vec{C}_1[^{proceed/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:

```
class c <: d { m [∅ ; b₁, b₂] }
advice b₁ (x̄) : exec (d :: m) { ĉ₁ }
advice b₂ (x̄) : exec (d :: m) { ĉ₂ }
```
is woven recursively as

```
class c <: ... { m (x̄) { ĉ₁[this/target, this [∅ ; b₂]/proceed] } }
advice b₂ (x̄) : exec (d :: m) { ĉ₂ }
```

- 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)

\[
P_{ABL} \xrightarrow{\text{weave}} P_{CBL} \quad \text{complete as} \quad P'_{ABL} \xrightarrow{\text{weave}} P'_{CBL}\]
The Full Untyped AOL

<table>
<thead>
<tr>
<th>Elements</th>
<th>Name</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a, \ldots, z$</td>
<td>Name</td>
<td>Command</td>
</tr>
<tr>
<td>$P, Q ::= (\overline{D} \vdash \overline{H})$</td>
<td>Program</td>
<td>New Declaration</td>
</tr>
<tr>
<td>$D, E ::= $</td>
<td>Declaration</td>
<td>Return</td>
</tr>
<tr>
<td>\hspace{1em}class $c &lt;:\overline{d} { \overline{M} }$</td>
<td>Class</td>
<td>Value</td>
</tr>
<tr>
<td>\hspace{1em}advice $a (\overline{x}) : \phi { \overline{C} }$</td>
<td>Advice</td>
<td>Get Field</td>
</tr>
<tr>
<td>$M ::= m [\overline{a} ; \overline{b}]$</td>
<td>Method</td>
<td>Set Field</td>
</tr>
<tr>
<td>$H, G ::= $</td>
<td>Heap Element</td>
<td>Static Message</td>
</tr>
<tr>
<td>\hspace{1em}object $o : c { \overline{F} }$</td>
<td>Object</td>
<td>Dynamic Message</td>
</tr>
<tr>
<td>\hspace{1em}thread $o { S }$</td>
<td>Thread</td>
<td>Advised Message</td>
</tr>
<tr>
<td>$F ::= f = v$</td>
<td>Field</td>
<td>Proceed</td>
</tr>
<tr>
<td>$S, T ::= $</td>
<td>Call Stack</td>
<td>Pointcut</td>
</tr>
<tr>
<td>\hspace{1em}\overline{C}$</td>
<td>Current Frame</td>
<td>False</td>
</tr>
<tr>
<td>\hspace{1em}let $x = o { S } ; \overline{C}$</td>
<td>Pushed Frame</td>
<td>Negation</td>
</tr>
</tbody>
</table>

| phi, psi ::= |
| false     |
| $\neg \phi$ |
| $\phi \lor \psi$ |

| $\phi, \psi ::= $ |
| call ($c :: m$) |
| exec ($c :: m$) |
Types (Unpublished)
A symptom: the following code compiles in AspectJ1.1.

```java
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:

  \[
  \text{class } c <: \ldots \{ m[\bar{a} ; \bar{b}] , \ldots \}
  \]

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

  \[
  \text{class } c <: \ldots \{ \text{protected } s \text{ method } m(\bar{t}) : r [\bar{a} ; \bar{b}] \ldots \}
  \]

- *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:

\[
\rho ::= \quad \text{Placement} \\
\quad \text{around} \quad \text{Around} \\
\quad \text{replace} \quad \text{Replace} \\
D, E ::= \quad \text{Declaration} \\
\rho \text{ advice } a (\vec{x} : \vec{t}) : r \text{ at } \phi \{ \vec{C} \} \quad \text{Advice}
\]
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

\[
a, \ldots, z
\]

\[
X, Y, Z ::= n : t
\]

\[
P, Q ::= (\bar{D} \vdash \bar{H})
\]

\[
\rho ::= \\
\quad \text{around} \\
\quad \text{replace}
\]

\[
D, E ::= \\
\quad \text{class } c <: d \{ \bar{F} \bar{M} \}
\]

\[
\rho \text{ advice } a (\bar{X}) : r \text{ at } \phi \{ \bar{C} \}
\]

\[
M ::= \text{protected } s \text{ method } m (\bar{i}) : r \text{ [}\bar{a} ; \bar{b}]\]

\[
F ::= \text{protected } s \text{ field } f : t;
\]

\[
V ::= f = v;
\]

\[
H, G ::= \\
\quad \text{object } o : c \{ \bar{V} \}
\]

\[
\text{thread } o \{ S \}
\]

\[
S, T ::= \\
\quad \bar{C}
\]

\[
\quad \text{let } X = o \{ S \}; \bar{C}
\]

\begin{align*}
\text{Name (\& Type)} & \quad C, B ::= \\
\text{Typed Name} & \quad \text{new } \bar{D} \bar{H}; \\
\text{Program} & \quad \text{return } v; \\
\text{Placement} & \quad \text{let } X = v; \\
\text{Around} & \quad \text{let } X = o.f; \\
\text{Replace} & \quad \text{set } o.f = v; \\
\text{Declaration} & \quad \text{let } X = o.c::m (\bar{v}); \\
\text{Class} & \quad \text{let } X = o.c.m (\bar{v}); \\
\text{Advice} & \quad \text{let } X = \text{proceed} (\bar{v}); \\
\text{Method} & \quad \phi, \psi ::= \\
\text{Field Type} & \quad \text{call } (c::m) \\
\text{Field Value} & \quad \text{exec } (c::m) \\
\text{Heap Element} & \quad \text{\neg call } (c::m) \\
\text{Object} & \quad \text{\neg exec } (c::m) \\
\text{Thread} & \quad \text{true} \\
\text{Current Frame} & \quad \text{false} \\
\text{Pushed Frame} & \quad \phi \wedge \psi \\
\text{Command} & \quad \phi \lor \psi
\end{align*}
\( P, Q, R ::= \)

\[
\begin{align*}
\text{let } x &= p \to q : \bar{m}; P \\
\text{return } \nu \\
\text{role } p < q; P \\
\text{advice } a[\phi] &= \sigma x . \tau y . \pi b . Q; P
\end{align*}
\]

\( \begin{array}{ll}
\text{Program} & \text{Message} \\
\text{Return} & \text{New Role} \\
\text{New Advice} & \end{array} \)
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 ($\sigma$).
- 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

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

Advice names are not first class.
# Pointcuts

## Syntax

\[ \phi, \psi ::= \text{Pointcut} \]

- \( p \rightarrow q : \ell \) \quad \text{Call}
- \( \neg p \rightarrow q : \ell \) \quad \text{Not Call}
- \( \phi \land \psi \mid \text{true} \) \quad \text{Conjunction}
- \( \phi \lor \psi \mid \text{false} \) \quad \text{Disjunction}
- \( \forall x \leq p . \phi \) \quad \text{Universal}
- \( \exists x \leq p . \phi \) \quad \text{Existential}

## Semantics

\[ \vec{D} \vdash p \leq q \]

\[ \vec{D} \vdash p \rightarrow q : \ell \ \text{sat} \ \phi \]
Dynamic Semantics

\[ \tilde{D}; \text{let } z = p \rightarrow q : \tilde{m}, \ell ; P \rightarrow \tilde{D}; \text{let } z = p \rightarrow q : \tilde{m}, \tilde{a} ; P \]

where \( \langle \tilde{a} \rangle = \langle a | \tilde{D} \ni \text{advice } a[\phi] \cdots \text{ and } \tilde{D} \vdash p \rightarrow q : \ell \text{ sat } \phi \rangle \)

\[ \tilde{D}; \text{let } z = p \rightarrow q : \tilde{m}, a ; P \rightarrow \tilde{D}; B[p/x, q/y, \tilde{m}/b] ; P[v/z] \]

where \( \tilde{D} \ni \text{advice } a[\cdots] = \sigma x \cdot \tau y \cdot \pi b \cdot \tilde{B} \); return \( v \)

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

Renaming required in second rule — \( \text{dom}(\tilde{B}) \) and \( \text{fn}(P) \) disjoint.

Garbage collection \( P^{\text{gc}} \rightarrow P' \) removes unused roles, advice, messages.
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 \( \ell \).
Call-by-value Lambda Calculus

\[ \vec{D} = \text{role } f; \]
\[
\text{advice } a[f . \text{call}] = \tau y . \text{let } x = y \rightarrow y : \text{arg}; P; \\
\text{role } g < f; \\
\text{advice } b[g . \text{arg}] = Q; \\
\]
\[
(\lambda x . P) Q \rightarrow \vec{D}; g \rightarrow g : \text{call} \\
\rightarrow \vec{D}; g \rightarrow g : a \\
\rightarrow \vec{D}; \text{let } x = g \rightarrow g : \text{arg}; P \\
\rightarrow \vec{D}; \text{let } x = g \rightarrow g : b; P \\
\rightarrow \vec{D}; \text{let } x = Q; P \\
\rightarrow \text{gc let } x = Q; P
\]

Cf. [Milner Functions as Processes]
Conditional

if $p \leq q$ then $R_1$ else $R_2 \triangleq$ role $r$;

advice $[\exists x \leq \text{top} . x \rightarrow r : \text{if}] = R_2$;

advice $[\exists x \leq q . x \rightarrow r : \text{if}] = R_1$;

$p \rightarrow r : \text{if}$

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

$\vec{D}; \text{if } p \leq q \text{ then } R_1 \text{ else } R_2 \xrightarrow*g \{ \begin{align*}
R_1 & \quad \text{if } \vec{D} \vdash p \leq q \\
R_2 & \quad \text{otherwise}
\end{align*} \}$
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 → Q} ≻ P` attaches *after* advice `λz . Q` to hook `p`.
- `{p . z → Q} ≪ P` attaches *before* advice `λz . Q` to hook `p`.
- `p⟨P⟩` evaluates `P` then runs advice hooked on `p`.

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

\[ P \triangleq \text{new } p; \{ p . x_1 \rightarrow x_1 + 1 \} \preceq \{ p . x_2 \rightarrow x_2 \ast 2 \} \Rightarrow p \langle 3 \rangle \]

\[ \vec{D} \triangleq \text{role } p; \]

\[ \text{advice } a[p . \text{hook}] = \lambda x_0 . x_0; \]

\[ \text{advice } b[p . \text{hook}] = \tau z . \pi d . \lambda x_1 . \text{let } y_1 = x_1 + 1; (z \rightarrow z : d)(y_1); \]

\[ \text{advice } c[p . \text{hook}] = \tau z . \pi d . \lambda y_2 . \text{let } x_2 = (z \rightarrow z : d)(y_2); x_2 \ast 2; \]

\[ P = \vec{D}; (p \rightarrow p : \text{hook}) 3 \]

\[ \rightarrow \vec{D}; (p \rightarrow p : a, b, c) 3 \]

\[ \rightarrow^{gc} \vec{D}; \text{let } x_2 = (p \rightarrow p : a, b)(3); x_2 \ast 2 \]

\[ \rightarrow^{gc} \vec{D}; \text{let } x_2 = (\text{let } y_1 = 3 + 1; (p \rightarrow p : a)(y_1)); x_2 \ast 2 \]

\[ \rightarrow^{gc} \vec{D}; \text{let } x_2 = (p \rightarrow p : a)(4); x_2 \ast 2 \]

\[ \rightarrow^{gc} \vec{D}; \text{let } x_2 = 4; x_2 \ast 2 \]

\[ \rightarrow^{gc} 8 \]
Translating the CBL

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

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

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

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

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

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

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

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

One step in CBL = Several steps in \(\mu\)ABC (including garbage collection).
Insight from $\mu$ABC

- Advice + Names + Name Substitution = Enough!
- Not much more complicated than $\lambda$, $\pi$ or $\sigma$.
- Paper includes spaghetti CPS translation of $\mu$ABC into $\pi$.
- 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].
  \[
  \begin{align*}
  \text{Stack} : & \text{Subject} \\
  \text{Policy} : & \text{Stack} \times \text{Object} \mapsto 2^{\text{Rights}}
  \end{align*}
  \]

- History-Based Access Control [Abadi Fournet 2003].
  \[
  \begin{align*}
  \text{Event} : & \text{Subject} \times \text{Object} \times \text{Value} \times \{\text{call, return}\} \\
  \text{History} : & \text{Event} \\
  \text{Policy} : & \text{History} \times \text{Object} \mapsto 2^{\text{Rights}}
  \end{align*}
  \]
// 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: \texttt{call}
- Stack Inspection: \texttt{call} + \texttt{cflow}
- History-Based: ?
A More General Notion of Past

- Connection between \texttt{cflow} and past-time eventuality operator $\Diamond$ has been noted by many.

- \texttt{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 \textit{abstraction} of the history.
A logic based on regular expressions and process algebraic operators:

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

Some encodings:

- balanced $= (\text{call};\text{return})!$
- semi \cdot balanced $= (\text{balanced};\text{call}^*)^*$
- cflow$\langle \phi \rangle = (\phi \land \text{call}^*) \parallel \text{balanced}$
Challenges for Temporal Pointcuts

- 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!