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.
I will try to say something interesting.

- Waffle.
- Limiting the power of AOP — Equational Reasoning
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
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)
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
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 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 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.
Fillman and Friedman: *The cleverness of classical AOP is augmenting conventional sequentiality with quantification, rather than supplanting it wholesale.*
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?
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

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

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

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

\[send(B,M)\] accepts \([M]\) and sends an \(ack\) to the source

\[send(B,M)\] accepts \([M]\) and blocks the call from Alice

\([M]\) is accepted and acknowledged by Bob

\([M]\) is blocked by Bob from going to Charlie

\(ack\) is acknowledged by the source

FOAL '04 – p.11/68
WWDD?

Are aspects the new goto?
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 lā AspectJ [Kiczales, et al])
- Blackbox AOP (a lā 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?
AOP in the Wild Wild West

AOP is exploring its power.

Wither formal aspects of aspects?

- Local sheriff — calls it like it is
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
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
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
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;
  }
}

java
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) {
    try {
      while (usedSlots == array.length) {
        wait();
      }
      catch (InterruptedException e) {}
    }
    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

<table>
<thead>
<tr>
<th>DJ</th>
<th>JAVA</th>
</tr>
</thead>
</table>
| ```java
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
    }
}
``` | ```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
    }
} ``` |
```
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 \_\_ networkNotFile
```
val fib = fn 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

```java
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

```java
// Methods for typechecking
public boolean Node.typeCheck() {
    /* ... */
}
public boolean WhileLoopNode.typeCheck() {
    /* ... */
}
public boolean IfThenNode.typeCheck() {
    /* ... */
}
```
Fig. 9. Some class definitions and their translation to composable mixins:  

```java
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);  
    }
}
/* Cannot merge for LockedShortDoor\textsuperscript{c} */

interface Door\textsuperscript{i} {  
    boolean canOpen(Person\textsuperscript{i} p);  
    boolean canPass(Person\textsuperscript{i} p);  
}
mixin Locked\textsuperscript{m} extends Door\textsuperscript{i} {  
    boolean canOpen(Person\textsuperscript{i} 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 Locked\textsuperscript{m} = Locked\textsuperscript{m}(Door\textsuperscript{c});
class Short\textsuperscript{m} = Short\textsuperscript{m}(Door\textsuperscript{c});
class LockedShort\textsuperscript{m} = Locked\textsuperscript{m}(Short\textsuperscript{m}(Door\textsuperscript{c}));
```
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*?
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]
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].
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].
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:

\[ \text{complete } P_{ABL} \xrightarrow{\text{weave}} P_{CBL} \]

as

\[ \text{complete } P'_{ABL} \xrightarrow{\text{weave}} P'_{CBL} \]

No reductions are gained:

\[ \text{complete } P_{ABL} \xrightarrow{\text{weave}} P_{CBL} \]

as

\[ \text{complete } P'_{ABL} \xrightarrow{\text{weave}} P'_{CBL} \]

(\[\rightarrow\] is OO reduction; \[\Rightarrow\] is AO reduction)
Example: $s$ delegates to $t$

```java
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.
```java
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();
}

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**

  \[
  \begin{align*}
  \text{object } o : c \{ ... f = v \ldots \} & \rightarrow \text{object } o : c \{ ... f = v \ldots \} \\
  \text{thread } \{ \text{let } x = o.f; \overline{C} \} & \rightarrow \text{thread } \{ \text{let } x = v; \overline{C} \}
  \end{align*}
  \]

- **Field set**

  \[
  \begin{align*}
  \text{object } o : c \{ ... f = u \ldots \} & \rightarrow \text{object } o : c \{ ... f = v \ldots \} \\
  \text{thread } \{ \text{set} o.f = v; \overline{C} \} & \rightarrow \text{thread } \{ \overline{C} \}
  \end{align*}
  \]

- **New declarations**

  \[
  \begin{align*}
  \text{thread } \{ \text{new class } c \leftarrow d \{ \ldots \} \}; \overline{C} \} & \rightarrow \text{class } c \leftarrow d \{ \ldots \} \\
  \text{object } o : c \{ \ldots \}; \overline{C} \} & \rightarrow \text{object } o : c \{ \ldots \} \\
  \text{thread } \{ \overline{C} \}
  \end{align*}
  \]
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, \%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 \( \vec{x} \) 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 \( x = \text{proceed} (\vec{v}) \); proceed to next advice
Supporting Call advice

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

        let x = o : c.m (v) ;

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

        thread p { S }

- These changes are required to implement the dynamic semantics.
Aspect Reduction: Context

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

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

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

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

object $o : d \{ ... \}$

class $d <: c \{ ... \}$

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

Actual type of $o$ is $d$.

Declared type of $o$ in thread is $c$. 
Aspect Reduction: Fetching Advice

advice \( a_0 \) (\( x \)) : 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 \( x = o :: c.m(v); \) \}

\[ \rightarrow \]

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

\[ \rightarrow \]

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

Controlling context is \( p \).
Aspect Reduction: Exec 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 $p \{ \text{let } x = o. [\emptyset ; b_1, b_2] (v) ; \}$

$\Rightarrow$

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

Controlling context is $o$. 
Encoding the CBL into the ABL

- Given a class:

  \[
  \text{class } c \leftarrow \text{Object } \{ \ldots m (\vec{x}) \{ \hat{C}_0 \} \ldots \} \\
  \text{class } d \leftarrow c \{ \ldots m (\vec{x}) \{ \hat{C}_1 \} \ldots \}
  \]

- Create exec advice for each body:

  \[
  \text{advice cbl}_c_m (\vec{x}) : \text{exec } (d \mathbin{::} m) \{ \hat{C}_0[^\text{proceed/\text{super}.m}] \}
  \]

  \[
  \text{advice cbl}_d_m (\vec{x}) : \text{exec } (d \mathbin{::} m) \{ \hat{C}_1[^\text{proceed/\text{super}.m}] \}
  \]

- 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:

  ```
  class c <: d { m[∅ ; b₁, b₂] }
  advice b₁ (⃗x) : exec (d :: m) { ⃗C₁ }
  advice b₂ (⃗x) : exec (d :: m) { ⃗C₂ }
  
  is woven recursively as

  class c <: ... { m (⃗x) { ⃗C₁[ths/target, ths. [∅ ; b₂]/proceed] } }
  advice b₂ (⃗x) : exec (d :: m) { ⃗C₂ }
  
  - 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)

\[ P_{ABL} \xrightarrow{\text{weave}} P_{CBL} \]

complete as

\[ P'_{ABL} \xrightarrow{\text{weave}} \sim P'_{CBL} \]
```
\begin{align*}
  a, \ldots, z \\
  P, Q & ::= (\vec{D} \vdash \vec{H}) \\
  D, E & ::= \\
  & \quad \text{class } c <: d \{ \vec{M} \} \\
  & \quad \text{advice } a (\vec{x}) : \phi \{ \vec{C} \} \\
  M & ::= m [\vec{a} ; \vec{b}] \\
  H, G & ::= \\
  & \quad \text{object } o : c \{ \vec{F} \} \\
  & \quad \text{thread } o \{ \vec{S} \} \\
  F & ::= f = v \\
  S, T & ::= \\
  & \quad \vec{C} \\
  & \quad \text{let } x = o \{ \vec{S} \} ; \vec{C} \\
\end{align*}
```

**Name**

- **Program**
- **Declaration**
- **Class**
- **Advice**
- **Method**
- **Heap Element**
- **Object**
- **Thread**
- **Field**
- **Call Stack**
- **Current Frame**
- **Pushed Frame**

**Command**

- New Declaration
- Return
- Value
- Get Field
- Set Field
- Static Message
- Dynamic Message
- Advised Message
- Proceed

**Pointcut**

- False
- Negation
- Disjunction
- Call
- Execution
Types (Unpublished)
Typing is Problematic

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:

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

- \texttt{protected} is “protected } c”; \texttt{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:

  $\rho ::= \quad \text{Placement}$
  
  - around
  - 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

\[ a, \ldots, z \]
\[ X, Y, Z ::= n : t \]
\[ P, Q ::= (\tilde{D} \vdash \tilde{H}) \]
\[ \rho ::= \]
\[ \quad \text{around} \]
\[ \quad \text{replace} \]
\[ D, E ::= \]
\[ \quad \text{class } c <: d \{ \tilde{F} \tilde{M} \} \]
\[ \quad \rho \text{ advice } a (\tilde{X}) : r \text{ at } \phi \{ \tilde{C} \} \]
\[ M ::= \text{protected } s \text{ method } m (\tilde{i}) : r [\tilde{a} ; \tilde{b}] \]
\[ F ::= \text{protected } s \text{ field } f : t; \]
\[ V ::= f = v; \]
\[ H, G ::= \]
\[ \quad \text{object } o : c \{ \tilde{V} \} \]
\[ \quad \text{thread } o \{ S \} \]
\[ S, T ::= \]
\[ \quad \tilde{C} \]
\[ \quad \text{let } X = o \{ S \}; \tilde{C} \]
\[ \text{Name (\& Type)} \]
\[ \text{Typed Name} \]
\[ \text{Program} \]
\[ \text{Placement} \]
\[ \quad \text{Around} \]
\[ \quad \text{Replace} \]
\[ \text{Declaration} \]
\[ \quad \text{Class} \]
\[ \quad \text{Advice} \]
\[ \text{Method} \]
\[ \text{Field Type} \]
\[ \text{Field Value} \]
\[ \text{Heap Element} \]
\[ \quad \text{Object} \]
\[ \quad \text{Thread} \]
\[ \text{Call Stack} \]
\[ \quad \text{Current Frame} \]
\[ \quad \text{Pushed Frame} \]
\[ \text{Command} \]
\[ \text{New} \]
\[ \text{Return} \]
\[ \text{Value} \]
\[ \text{Get Field} \]
\[ \text{Set Field} \]
\[ \text{Static Message} \]
\[ \text{Dynamic Msg} \]
\[ \text{Advised Msg} \]
\[ \text{Proceed} \]
\[ \text{Pointcut} \]
\[ \text{Call} \]
\[ \text{Execution} \]
\[ \text{Not Call} \]
\[ \text{Not Execution} \]
\[ \text{True} \]
\[ \text{False} \]
\[ \text{Conjunction} \]
\[ \text{Disjunction} \]
\[ P, Q, R ::= \]
\[ \text{let } x = p \rightarrow q : \vec{m}; P \]
\[ \text{return } v \]
\[ \text{role } p < q; P \]
\[ \text{advice } a[\phi] = \sigma x . \tau y . \pi b . Q; P \]
A Minimal Aspect-Based Calculus
Design Choices

Goals

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

Goals

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

Decisions

- Start with Abadi and Cardelli’s object calculus ($\sigma$).
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).
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.
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

\[ f, \ldots, \ell, p, \ldots, z \]

Label or Role

\[ a, \ldots, e \]

Advice name

\[ m, n ::= \ell \mid a \]

Message

\[ P, Q ::= \vec{B}; \text{return } v \]

Program

\[ B, C ::= \text{let } x = p \rightarrow q : \vec{m} \mid D \]

Command

\[ D, E ::= \]

Declaration

\[ \text{role } p < q \]

Role

\[ \text{advice } a[\phi] = \sigma x . \tau y . \pi b . Q \]

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

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

\[ \vec{D} \vdash p \rightarrow q : \ell \ \text{sat} \ \phi \]
\[ \bar{D}; \text{let } z = p \rightarrow q : \bar{m}, \ell; P \rightarrow \bar{D}; \text{let } z = p \rightarrow q : \bar{m}, \bar{a}; P \]

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

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

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

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

Renaming required in second rule — \( \text{dom}(\bar{B}) \) and \( \text{fn}(P) \) disjoint.
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 \( \langle \tilde{a} \rangle = \langle a \mid \tilde{D} \ni \text{advice } a[\phi] \cdots \text{ and } \tilde{D} \vdash p \rightarrow q : \ell \text{ sat } \phi \rangle \)

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

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

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 \( P \xrightarrow{\text{gc}} P' \) 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

\[
\vec{D} = \text{role } \vec{f};
\]

advice \(a[f . \text{call}] = \tau y. \text{let } x = y \rightarrow y : \text{arg}; P\);

role \(g < \vec{f};\)

advice \(b[g . \text{arg}] = Q;\)

\[
(\lambda x . P) Q \rightarrow \vec{D}; g \rightarrow g : \text{call}
\]

\[\rightarrow \vec{D}; g \rightarrow g : a\]

\[\rightarrow \vec{D}; \text{let } x = g \rightarrow g : \text{arg}; P\]

\[\rightarrow \vec{D}; \text{let } x = g \rightarrow g : b; P\]

\[\rightarrow \vec{D}; \text{let } x = Q; P\]

\[\text{gc} \rightarrow \text{let } x = Q; P\]

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.

\[ \vec{D} ; \text{if } p \leq q \text{ then } R_1 \text{ else } R_2 \rightarrow^{* \rightarrow} \begin{cases} R_1 & \text{if } \vec{D} \vdash p \leq q \\ R_2 & \text{otherwise} \end{cases} \]
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.
- $\{ p . z \rightarrow Q \} \triangleright P$ attaches after advice $\lambda z . Q$ to hook $p$.
- $\{ p . z \rightarrow Q \} \triangleleft P$ 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

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

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

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

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

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

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

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

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

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

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

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

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

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

One step in CBL = Several steps in \(\mu\)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].
  
  Policy: Subject $\times$ Object $\mapsto 2^{\text{Rights}}$

- Stack Inspection [Wallach et al 1997].
  
  Stack: Subject
  
  Policy: Stack $\times$ Object $\mapsto 2^{\text{Rights}}$

- History-Based Access Control [Abadi Fournet 2003].
  
  Event: Subject $\times$ Object $\times$ Value $\times \{\text{call, return}\}$
  
  History: Event
  
  Policy: History $\times$ Object $\mapsto 2^{\text{Rights}}$
// 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 \textit{cflow} and past-time eventuality operator has been noted by many.

- \textit{cflow}'s limitations are accepted on grounds of implementability.

How can we implement a more general notion of past?
A More General Notion of Past

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

- \textit{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.
Sketching a Logic of Temporal Pointcuts

A logic based on regular expressions and process algebraic operators:

\[ \epsilon \quad \text{empty.} \]
\[ \phi; \psi \quad \text{sequential composition of two traces.} \]
\[ \phi^* \quad \text{closure of sequential composition} = \epsilon \lor (\phi; \phi^*). \]
\[ \phi \parallel \psi \quad \text{parallel composition of two traces.} \]
\[ \phi! \quad \text{closure of parallel composition} = \epsilon \lor (\phi \parallel \phi!). \]

Some encodings:

\[ \text{balanced} = (\text{call}; \text{return})! \]
\[ \text{semi} \cdot \text{balanced} = (\text{balanced}; \text{call}^*)^* \]
\[ \text{cflow} \langle \phi \rangle = (\phi \land \text{call}^*) \parallel \text{balanced} \]
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!