Continuation Models for AOP

Christopher Dutchyn

University of Saskatchewan
Software Research Lab
How Might We Program Display Updating?

```java
class Point extends Shape {
    private int x = 0, y = 0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
}
```

```java
class Line extends Shape {
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
}
```
Object-Oriented Solution

- Separate declaration of behaviour for operations
  - setX
  - setY
  - setP1
  - setP2

- Each operation does its own thing

- Each operation
  - updates display in a way
  - consistent with others

```java
class Point extends Shape {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) { this.x = x;
                      Display.update(); }
    void setY(int y) { this.y = y;
                      Display.update(); }
}

class Line extends Shape {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) { this.p1 = p1;
                         Display.update(); }
    void setP2(Point p2) { this.p2 = p2;
                         Display.update(); }
}
```
Same Behaviour ... Different Modularity

• Aspect declares
  – Some points in execution represent a display state change
    • execution of methods matching this pattern
  – After a change occurs
    • update the display
Simple Comparison

AO Solution

• Display updating is modularized into a single location

• Behaviour of
  – each shape is manifest in single module
  – display updating is manifest in single module

• Interaction between display updating and shape movement is explicit

OO Solution

• Display updating is
  – scattered across multiple data modules
  – tangled with the code in those modules

• Behaviour of each shape and associated display updating is manifest in single module

• Interaction between display updating and each shape's movement is explicit
AOP Provides a New Kind of Modularity

class Point extends Shape {
    private int x = 0, y = 0;
    int getX() { return x; }  
    int getY() { return y; }
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
}

class Line extends Shape {
    private Point p1, p2;
    Point getP1() { return p1; }  
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
}

aspect DisplayUpdating {
    pointcut change(): execution(void Shape+.set*());
    after() returning: change() {
        Display.update();
    }
}
AOP Provides a New Kind of Modularity

```java
class Point extends Shape {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
}

class Line extends Shape {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
}

aspect DisplayUpdating {
    pointcut change(): execution(void Shape+.set*(*));
    after() returning: change() {
        Display.update();
    }
}
```
AOP Provides a New Kind of Modularity

```java
class Point extends Shape {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
}

class Line extends Shape {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
}

aspect DisplayUpdating {
    pointcut change(): execution(void Shape+.set*(*));

    after() returning: change() {
        Display.update();
    }
}
```
AOP Provides a New Kind of Modularity

class Point extends Shape {
    private int x = 0, y = 0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
}

class Line extends Shape {
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
}

aspect DisplayUpdating {
    pointcut change(): execution(void Shape+.set*(*));
    after() returning: change() {
        Display.update();
    }
}
Join Points, Pointcuts, and Advice

An Intellectual Model of AOP
Without Continuations

```
(define (pick x)
  (if x 1 2))
(+ 3 (pick #t))
```

```
add 3 and 1
```
Without Continuations

```
(define (pick x)
  (if x 1 2))
(+ 3 (pick #t))
```

```
add 3 and 1
```
Continuations

[Strachey+ ’74; Reynolds ’74; Meyers ’85; …]

- Continuations reify control state
  - Escape semantics ⇒ not composable

(define (pick x) (if x 1 2))
(+ 3 (pick #t))

Evaluation of operands has continuation

\[
\text{(evlis (pick #t) p) (λ(f b) (eval (body f) (extend (env f) (id f) b) (λ(i) (+ 3 i))))}
\]
Sub-Continuations [Felleisen ’88; Hieb+ ’94; Shan ’02; Agere+’05; ...]

- Structure within continuations
  - composable

```
(define (pick x) (if x 1 2))
(+ 3 (pick #t))
```

```
(evlis (pick #t) ρ (push [exec-proc] κ))
```

```
⇒ (⟨CLO⟩ #t) ⇔ (λ(f b) (eval (body f)
  (extend (env fun) (id fun) b)
  κ))
```

```
⇒ 1  ⇔ (λ(i) (+ 3 i)))
```

```
⇒ 4
```

execution sub-continuation

evaluation of operands has continuation

DCC Chile
Join Points Modeled by Sub-Continuations

- “Principled points in execution”
  - join points correspond to sub-continuations

- call join point ≡ dispatch sub-continuation

  (send aPoint setX 7)

  (evlis (aPoint,7) ρ (push \text{\{dispatch setX\}} κ))

  ...⇒ (⟨obj⟩,7)⇒ (λ(o v) (apply (dispatch o setX)

  (push \text{\{exec-method o v\}} κ))

  ⇒ ⟨METH⟩⇒ (λ(m) (apply-method m o v κ))

  ⇒ (eval (body m) [this→o (ids m)→v] κ)

  ...

- execution join point ≡ exec-method sub-continuation

- field get/set join points …
Procedures Transform Continuations [Filinski ’89; Griffin ’91; Murthi ’92]

- Procedures have two different modes of application:
  - Applied to a value: they yield another value
  - Applied to a continuation: they yield another continuation

\[
\text{(define (pick x) (if x 1 2))}
\]
\[
(+ 3 (\text{pick } \text{#t} ))
\]

Transforms \textit{value}

\[
\downarrow
\]

\text{Takes } \lambda(i) (+ 3 i) \text{ to } \lambda(b) ((\lambda(i) (+ 3 i)) (\text{if } b 1 2))

\text{Transforms \textit{continuation}}

\[
(+ 3 1)
\]

\text{Takes \#t to 1}

\text{pick :: Bool \rightarrow Int}

\text{pick :: ¬Int \rightarrow ¬Bool}
Advice Modeled as Sub-Cont Transformers

- Advice body extends sub-continuation behaviour

(pointcut change (execution (Point setX)))

(around change (λ(o v)
  (proceed o v)
  (send display update o)))

(send aPoint setX 7)

(evlis (aPoint,7) ρ (push (advise 〈dispatch setX〉 ) κ))

⇒ 〈obj〉,7) ⇔ (λ(o v) (apply (dispatch o setX)
  (push (advise 〈exec-method o v〉 ) κ))

⇒ 〈METH〉 ⇔ (λ(m) (apply-advice ADV m o v κ))

⇒ (eval (body ADV) [o→o v→v proceed→(λ(o v κ) (apply-method m o v κ))] κ)

⇒ (eval 〈proceed o v〉 [...] (push 〈next 〈send display update o〉 [...] κ))

⇒ (apply-method m o v (push 〈next 〈send display update o〉 [...] κ))

⇒ ? ⇔ (λ(_) (eval (send display update o) [...] κ))

⇒ ? ⇔ κ
Pointcuts Modeled as Sub-Cont Identifiers

• Pointcuts match sub-continuation structures

\[(\text{pointcut change (execution (Point setX)))}\]

\(\text{(around change (λ(o v))})
\text{(proceed o v))}
\text{(send display update o)))}\]

\(\)\(\)\(\)\(\)\(\)

\text{(send aPoint setX 7)}
\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)

\(\)\(\)\(\)
Join Points Modeled by Sub-Conts

• “Principled points in execution”
  – join points correspond to sub-continuations

  – call join point $\equiv$ dispatch sub-continuation

  $(\text{send} \, \text{aPoint} \, \text{setX} \, 7)$

  $(\text{evlis} \, (\text{aPoint},7) \, \rho \, (\text{push} \, \langle \text{dispatch} \, \text{setX} \rangle \, \kappa))$

  $\Rightarrow (\langle \text{obj} \rangle,7) \mapsto (\lambda(o \, v) \, (\text{apply} \, (\text{dispatch} \, o \, \text{setX})$

  $(\text{push} \, \langle \text{exec-method} \, o \, v \rangle \, \kappa))$

  $\Rightarrow \langle \text{METH} \rangle \mapsto (\lambda(m) \, (\text{eval} \, (\text{body} \, m) \, [\text{this} \rightarrow o \, (\text{ids} \, m) \rightarrow v] \, \kappa))$

  $\Rightarrow \ldots$

  – execution join point $\equiv$ execution sub-continuation

  – field get/set join points …
Structuring: Applicability Determines Proceed

Default behaviour

Some call join points

More call join points

```
aspect DisplayUpdating {
  pointcut change(Shape s):
    this(shape) && execution(void Shape+.set(""));
  after(Shape s): change(s) {
    shape.display.update(s);
  }
  pointcut smallerChange(Shape s) :
    change(s) && cflow(execution(void UI.addShape()));
  after(Shape s) : smallerChange(s) {
    System.error.println("Shape added, displayed.");
  }
}
```
Model Abstracts Computations

- Well-founded in prog. language theory
  - Join points $\equiv$ sub-continuation
  - Advice $\equiv$ procedure-like transform to join point
  - Pointcuts $\equiv$ sub-continuation identifiers

- Abstraction: Control
  - Pointcuts identify join points
    - computations delimited by continuations

- Interface: Extension/Replacement
  - Advice captures those computations and
    - extends/replaces those computations
  - altering their control structure
AOP Provides a New Kind of Modularity

```java
class Point extends Shape {
    private int x = 0, y = 0;
    int getX() { return x; }  
    int getY() { return y; }
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
}

class Line extends Shape {
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
}
```

**pointcut**  
≡ kinds of computations  

**join point**  
≡ computation  

**aspect DisplayUpdating**  

```java
aspect DisplayUpdating {
    pointcut change():
        execution(void Shape+.set*(*));
    after() returning: change() {
        Display.update();
    }
}
```  

**replace/extend**  

**advice**  
≡ computation transformation
Characterizing Control
Values are Characterized by Types [Cousot ’97; Pierce ’02]

- **Int**
  - 32-bit 2’s-complement
  - Primitives
    - (printf “%d” ...)
    - +, -, *, /, =
  - Passed as argument

- **Bool → Int**
  - closures
  - Application
  - Passed as argument

**Static checking**

- Safety
- Machine-checked compliance to annotated intent
- Enables optimizations
Join Points Carry Effects

- **Exceptions**
  - May throw division by zero
- **State**
  - Reads value
  - Mutates value
- **Input/Output**
  - Reads file
  - Writes file
- **Concurrency**
  - Generates new thread
  - Blocks on visible thread
- **Sequencing**
- **Non-determinism**
- **Partiality**

- **Walk the AST and determine**
  - Throw/Catch
  - Read
  - Display
  - SetField
  - GetField
  - Fork
  - Exit
  - Wait
  - Used in the join point *shadows*

- **Can determine effect type of join point shadows**
Pointcuts Have Merged Join Point Effect Type

• Merger provides opportunity to examine types

(pointcut change (or (execution (Point setX))
(execution (Line setP1)))))

• Check
  – Excluded join points with similar effect type?
    • Were these ones missed?
  – All the join points have same effect type, except one?
    • Was this one accidentally included?
Advice Composes Additional Effects

Logging: I/O

(pointcut action ...)  
(around action (λ args  
  (write "before: ... ")  
  (apply proceed args)))

Transaction: remove state

(pointcut update ...)  
(around update (λ args  
  (let ([saved (get-state)])  
    (with-handlers (λ(exc)  
      (rollback-state saved))  
      (apply proceed args))))))

Asynchronous: add concurrency

(pointcut operation ...)  
(around operation (λ args  
  (if (fork)  
    (apply proceed args))))
Effects Compose in Layers

[Jones+ 96; Filinski ’99; …]

- Exceptions over State
  \[ Ta = (1,s)+(a,s) \]
  \( \Rightarrow \) Transaction

- State over Concurrency
  \[ Ta = [a],s \]
  \( \Rightarrow \) Global store
  \( \Rightarrow \) Atomicity is a potential problem

- Concurrency over State
  \[ Ta = [(a,s)] \]
  \( \Rightarrow \) Thread-local store

- Java
  \[ Ta = [(1+a),s_{\text{local}},s_{\text{global}}] \]
  - Exceptions over state over concurrency over state
Some Compositions are Wrong

• Examples
  – Transaction over IO
    • output cannot be undone
    • some input cannot be undone
  – Transaction over Concurrency
    • Concurrent operations may see incomplete transaction

• Effect checking summarizes behaviour
  – Enables identifying inconsistent interactions
Some Compositions are Potentially Wrong

• Examples:
  – Concurrency and State
    • Either order is valid – but which is desired?
      – Thread-local state
      – Shared state
  – Concurrency over IO
    • With shared communication channels, reads and writes can interfere

• The programmer knows the intent and needs to decide
  – We can provide report to locate trouble spots
Some Correct Interactions may be Flagged Wrong

- Example:
  - Logging in a Transaction
    - Stderr output in a transactional context?

- The checker complains
  - This is called slack in a type system
  - Need some work-around
AOP Provides a New Kind of Modularity

class Point extends Shape {
    private int x = 0, y = 0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
}

class Line extends Shape {
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
}

aspect DisplayUpdating {
    pointcut change():
    execution(void Shape+.set*(*));
    after() returning: change() {
        Display.update();
    }
}
Advice Description is Informative

```
(class Point Shape
  (field x)

  (class Line Shape
    (field p1)
    (field p2)

      (method getP1 () p1)
      (method getP2 () p2)

      (method setP1 (p) (field-set p1 p))
      (method setP2 (p) (field-set p2 p))
    )

  )

(pointcut change (or (execution (Point setX))
                     (execution (Point setY))
                     (execution (Line setP1))
                     (execution (Line setP2)))

  (around change (λ(o v)
                  (proceed o v)
                  (send display update o)))))
```

change: mutates(receiver field x)
  or mutates(receiver field y)
  or mutates(receiver field p1)
  or mutates(receiver field p2)

advice: sequence after
  input/output(file: stdout)
Advice Description is Informative

- Observes state changes
  - Each join point mutates object-local state
  - Pointcut abstracts local-state changes only

- Augments state changes
  - Adds IO effect to join point behaviour
  - Single unconditional proceed maintains existing sequential control flow

- Advice unconditionally couples shape state mutation with display state updating

```
(class Point Shape
  (field x)

(class Line Shape
  (field p1)
  (field p2)

  (method getP1 () p1)
  (method getP2 () p2)

  (method setP1 (p) (field-set p1 p))
  (method setP2 (p) (field-set p2 p))
)

(pointcut change (or (execution (Point setX)))
  (execution (Point setY))
  (execution (Line setP1))
  (execution (Line setP2)))

(around change (λ(o v)
  (proceed o v)
  (send display update o))))
```
Effect Typing for Aspects

• Provides summary report of behaviour of
  – join point shadows
  – point cuts
  – advice

• Developer can use reports to find
  – Anomalous join point shadows in pointcuts
  – Understand composed behaviour of
    • join point
    • advice
Related Work

- **[Rinard ’04]**
  - weaves AspectJ code then checks
  - applies pluggable data-flow and control-flow analyses

- **MiniMAO [Clifton ’05]**
  - distinguish two categories
  - recommend ‘surround’ to syntactically denote simple case

- **[Sihman+ ’03]**
  - distinguishes three categories
  - model-checking
Summary
AOP Provides Modularity over Control

```java
class Point extends Shape {
    private int x = 0, y = 0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
}

class Line extends Shape {
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
}
```

pointcut ≡ kinds of computations

aspect DisplayUpdating {
    pointcut change():
        execution(void Shape+.set*(*));
    after() returning: change() {
        Display.update();
    }
}

join point ≡ computation

replace/extend

advice ≡ computation transformation

restructures
Effect Typing Helps Understand Composition

class Point extends Shape {
    private int x = 0, y = 0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
}

class Line extends Shape {
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
}

aspect DisplayUpdating {
    pointcut change():
        execution(void Shape+.set*(*));
    after() returning: change() {
        Display.update();
    }
}
Contribution: Semantic Model

• Shows how AOP fits naturally within PL theory
  – No separate artifact required
    • meta-programs
    • weavers

• Subsumes other models:
  • first-class context labels [Dantas+ '04]
  • continuation marks [Dutchyn+ '06]
  • weavers [Wand+ '04; Bruns+ '04; Masuhara+ '03; Clifton '05]
  • predicate dispatch [Orleans '05]

• Clarifies AOP ⇔ reflective meta-programming
... What’s Missing?

- **Intertype declarations**
  - Join points exist in elaboration phase
    - Declare operation
    - Override implementation
    - Create class

- **Cflow**
  - Makes obvious that cflow adds state and breaks tail calls
  - Build as a sub-aspect construction

- **Other meta-programming AOP systems (hyperJ, composeJ)**
  - Given a precise dynamic semantics
    - Identifying sub-continuations is mechanical
  - Our construction goes through
Future Work

- Dynamic aspects modularize control
  - And associated operations
    - Just like objects modularize data
  - And associated operations

![Diagram of Frame Activation, Pointcut, AspectJ](image)

- Category theory?
Future Work

- Tantalizing aspects ↔ classes duality

<table>
<thead>
<tr>
<th></th>
<th>OO</th>
<th>AO</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base</strong></td>
<td>Value</td>
<td>Continuation</td>
</tr>
<tr>
<td></td>
<td>↓ product</td>
<td>↓ sum</td>
</tr>
<tr>
<td><strong>Bundle</strong></td>
<td>Object</td>
<td>Instance</td>
</tr>
<tr>
<td><strong>Abstract</strong></td>
<td>Class</td>
<td>Aspect</td>
</tr>
<tr>
<td></td>
<td>↓ sum</td>
<td>↓ product</td>
</tr>
<tr>
<td><strong>Structure</strong></td>
<td>Inheritance</td>
<td>?</td>
</tr>
</tbody>
</table>

• Gives framework for understanding the kinds of manipulations that AOP enables

<table>
<thead>
<tr>
<th></th>
<th>OO</th>
<th>AO</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dispatch</strong></td>
<td>Method</td>
<td>Constructor</td>
</tr>
<tr>
<td><strong>Order</strong></td>
<td>Most-to-least specific</td>
<td>Most-to-least applicable</td>
</tr>
<tr>
<td><strong>Static</strong></td>
<td>Super</td>
<td>Proceed</td>
</tr>
<tr>
<td><strong>Structure</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Future Work

• What annotations can scale-up aspect checking?
  – Showed tractable
  – Want practical
  – AspectJ?

• What optimizations can aspect effect-checking enable?
  – Related to effect hierarchy [Tolmach ’04]

• What about other effect taxonomies? [Thielecke ’04]
Future Work

aspect Barrier {
  private final int lastN = ...;
  private List<Thread> waiting = new ...;

  pointcut syncAfter(): ...;

  ... around(): syncAfter() {
    ... result = proceed();
    if (waiting.size() == lastN) {
      for (Thread t : waiting) { t.notify(); }
      waiting.clear();
    } else {
      Thread t = Thread.currentThread();
      waiting.add(t);
      t.wait();
    }
    return result
  }
}

aspect ThreadSafety {
  boolean isSafeThread();
  private Queue<Runnable> q = new ...;

  pointcut unsafeOperation(): ...;

  void around(): unsafeOperation() {
    if (isSafeThread()) {
      for (Runnable r : q) { r.run(); }
      proceed();
    } else {
      q.add(new Runnable() {
        public void run() {
          proceed();
        }
      });
    }
  }
}

aspect Asynchronous {
  pointcut operation(): ...;

  void around(): operation() {
    new Thread(new Runnable() {
      public void run() {
        proceed();
      }
    }).run();
  }
}

aspect Logging {
  pointcut action(): ...;

  ... around(): action() {
    System.out.println("before: ...");
    return proceed();
  }
}

aspect Transaction {
  private Object savedState;

  pointcut update(): ...;

  boolean around(): update() {
    savedState = getState();
    try {
      return proceed();
    } catch (Exception e) {
      rollBackTo(savedState);
      return false;
    }
  }
}
Power of the Abstraction

• Cω : C# + join calculus

• Their additions can be characterized by two abstract aspects
  – Asynchronicity
  – Barriers (Chords)

• Aspects are more general and more expressive

```java
aspect Barrier {
  private final int lastN = ...;
  private List<Thread> waiting = new ...;

  pointcut syncAfter(): ...;

  ... around(): syncAfter() {
    ... result = proceed ();
    if (waiting.size() == lastN) {
      for (Thread t : waiting) { t.notify(); }
      waiting.clear();
    } else {
      Thread t = Thread.currentThread();
      waiting.add(t);
      t.wait();
    }
    return result;
  }
}

aspect Asynchronous {
  pointcut operation(): ...;

  void around(): operation() {
    new Thread(new Runnable() {
      public void run() {
        proceed(); }
    }).run();

  }
}
```
Discussion

Questions?
Supporting Slides
Other Analyses – Rinard+

- [*Rinard ’04*] weaves AspectJ code, then checks
  - DFA identifies state interactions
    - **Orthogonal** \(\equiv\) aspect and base have independent state
    - **Independent** \(\equiv\) aspect doesn’t read base mutable state
    - **Observational** \(\equiv\) aspect reads base mutable state
    - **Actuation** \(\equiv\) aspect writes into base immutable state
    - **Interference** \(\equiv\) both write into each others state
Other Analyses – Rinard+

- [Rinard ’04] weaves AspectJ code, then checks
  - CFA identifies control interactions
    - **Augmentative** ≡ state effect, always proceeds
    - **Narrowing** ≡ conditional single proceed
    - **Replacement** ≡ unconditional no proceed
    - **Combinational** ≡ all other
Other Analyses – Clifton+, Katz+

- **[Clifton ’05]** MiniMOA distinguishes
  - *Spectators* ~ observational and augmentative
    - Can be ignored for (some) code understanding
  - *Assistants* ~ all else
    - Require them to documented in the affected module

- **[Katz+ ’04]** model-checks woven code to identify
  - *Spectative* ~ observational and augmentative
  - *Regulative* ~ observational and narrowing/replacement
  - *Invasive* ~ interference and/or combinational
The End

Really!

University of Saskatchewan
Software Research Lab