AOP and the Antinomy of the Liar

F. Forster F. Steimann

Programming Systems
Department of Computer Science
University of Hagen

FOAL 2006

Outline

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true)

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is false

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is false
2. false("Theorem One" is false)

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is false
2. false("Theorem One" is false) $\Rightarrow$ "Theorem One" is true

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is false
2. false("Theorem One" is false) $\Rightarrow$ "Theorem One" is true
3. true("Theorem Two" is true)

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true) ⇒ "Theorem Two" is false
2. false("Theorem One" is false) ⇒ "Theorem One" is true
3. true("Theorem Two" is true) ⇒ "Theorem Two" is true

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is false
2. false("Theorem One" is false) $\Rightarrow$ "Theorem One" is true
3. true("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is true
4. true("Theorem One" is false)

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true) ⇒ "Theorem Two" is false
2. false("Theorem One" is false) ⇒ "Theorem One" is true
3. true("Theorem Two" is true) ⇒ "Theorem Two" is true
4. true("Theorem One" is false) ⇒ "Theorem One" is false

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is false
2. false("Theorem One" is false) $\Rightarrow$ "Theorem One" is true
3. true("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is true
4. true("Theorem One" is false) $\Rightarrow$ "Theorem One" is false
5. false("Theorem Two" is true)

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is false
2. false("Theorem One" is false) $\Rightarrow$ "Theorem One" is true
3. true("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is true
4. true("Theorem One" is false) $\Rightarrow$ "Theorem One" is false
5. false("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is false

Famous Antinomies - 1

Theorem One

"Theorem Two" is true

Theorem Two

"Theorem One" is false

Interpretation

1. false("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is false
2. false("Theorem One" is false) $\Rightarrow$ "Theorem One" is true
3. true("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is true
4. true("Theorem One" is false) $\Rightarrow$ "Theorem One" is false
5. false("Theorem Two" is true) $\Rightarrow$ "Theorem Two" is false
6. repeat indefinitetly.

Famous Antinomies - 2

Aspect One

```
public aspect S1 {
  void around(): adviceexecution() && within(S2) {
    proceed();
  }
}
```

Aspect Two

```
public aspect S2 {
  void around(): adviceexecution() && within(S1) {
  }
}
```

Famous Antinomies - 3

Russell's Example

The Class of all those classes which are not members of themselves.

Famous Antinomies - 3

Russell's Example

The Class of all those classes which are not members of themselves.

$$M = \{X | X \notin X\}$$

Famous Antinomies - 3

Russell's Example

The Class of all those classes which are not members of themselves.

$$M = \{X | X \notin X\}$$

Assuming $M \in M$

$M \in M$ contradicts the characteristic function of M

Famous Antinomies - 3

Russell's Example

The Class of all those classes which are not members of themselves.

$$M = \{X | X \notin X\}$$

Assuming $M \in M$

$M \in M$ contradicts the characteristic function of $M \Rightarrow M \notin M$

Famous Antinomies - 3

Russell's Example

The Class of all those classes which are not members of themselves.

$$M = \{X | X \notin X\}$$

Assuming $M \in M$

$M \in M$ contradicts the characteristic function of $M \Rightarrow M \notin M$

Assuming $M \notin M$

$M \notin M$ fullfils the characteristic function of M

Famous Antinomies - 3

Russell's Example

The Class of all those classes which are not members of themselves.

$$M = \{X | X \notin X\}$$

Assuming $M \in M$

$M \in M$ contradicts the characteristic function of $M \Rightarrow M \notin M$

Assuming $M \notin M$

$M \notin M$ fullfils the characteristic function of $M \Rightarrow M \in M$

Great Escapes - Russell

Russell's Solution

Whatever involves all of a collection must not be one of the collection.

Great Escapes - Russell

Russell's Solution

Whatever involves all of a collection must not be one of the collection.

Theory of Types

Whatever contains a variable must not be a possible value of that variable.

Great Escapes - Russell

Russell's Solution

Whatever involves all of a collection must not be one of the collection.

Theory of Types

Whatever contains a variable must not be a possible value of that variable.

Theory of Types for AspectJ

$someadvice_1(\{jp_1, jp_2, \ldots\}) \Rightarrow joinpoints\ in\ someadvice_1 \notin \{jp_1, jp_2, \ldots\}$

Tracing - 1

Task

Trace all Methodexecution and Adviceexecution.

Tracing - 1

Task

Trace all Methodexecution and Adviceexecution.

First Go

```
public aspect Tracing {
  void around(): adviceexecution()||execution (* *(..)){
    System.out.println("Entering:" + thisJoinPoint);
    proceed();
    System.out.println("Leaving: " + thisJoinPoint);
  }
}
```

Tracing - 1

Task

Trace all Methodexecution and Adviceexecution.

First Go

```
public aspect Tracing {
  void around(): adviceexecution()||execution (* *(..)){
    System.out.println("Entering:" + thisJoinPoint);
    proceed();
    System.out.println("Leaving: " + thisJoinPoint);
  }
}
```

Problem

`around` `advice` is an *adviceexecution*() *joinpoint*!

Tracing - 1

Task

Trace all Methodexecution and Adviceexecution.

First Go

```
public aspect Tracing {
  void around(): adviceexecution()||execution (* *(..)){
    System.out.println("Entering:" + thisJoinPoint);
    proceed();
    System.out.println("Leaving: " + thisJoinPoint);
  }
}
```

Problem

`around` `advice` is an *adviceexecution*() *joinpoint*! ϟTheory of Types

Tracing - 2

Workaround

```
public aspect Tracing {
  pointcut guard(): (adviceexecution()
    ||execution (* *(..))) && within(Tracing);
  void around(): (adviceexecution()
    ||execution (* *(..))) && !cflow(guard()) {
    System.out.println("Entering:" + thisJoinPoint);
    proceed();
    System.out.println("Leaving: " + thisJoinPoint);
  }
}
```

Tracing - 2

Workaround

```
public aspect Tracing {
  pointcut guard(): (adviceexecution()
    ||execution (* *(..))) && within(Tracing);
  void around(): (adviceexecution()
    ||execution (* *(..))) && !cflow(guard()) {
    System.out.println("Entering:" + thisJoinPoint);
    proceed();
    System.out.println("Leaving: " + thisJoinPoint);
  }
}
```

Problem

Very verbose,

Tracing - 2

Workaround

```
public aspect Tracing {
  pointcut guard(): (adviceexecution()
    ||execution (* *(..))) && within(Tracing);
  void around(): (adviceexecution()
    ||execution (* *(..))) && !cflow(guard()) {
    System.out.println("Entering:" + thisJoinPoint);
    proceed();
    System.out.println("Leaving: " + thisJoinPoint);
  }
}
```

Problem

Very verbose, error-prone,

Tracing - 2

Workaround

```
public aspect Tracing {
  pointcut guard(): (adviceexecution()
    ||execution (* *(..))) && within(Tracing);
  void around(): (adviceexecution()
    ||execution (* *(..))) && !cflow(guard()) {
    System.out.println("Entering:" + thisJoinPoint);
    proceed();
    System.out.println("Leaving: " + thisJoinPoint);
  }
}
```

Problem

Very verbose, error-prone, redundancy,

Tracing - 2

Workaround

```
public aspect Tracing {
  pointcut guard(): (adviceexecution()
    ||execution (* *(..))) && within(Tracing);
  void around(): (adviceexecution()
    ||execution (* *(..))) && !cflow(guard()) {
    System.out.println("Entering:" + thisJoinPoint);
    proceed();
    System.out.println("Leaving: " + thisJoinPoint);
  }
}
```

Problem

Very verbose, error-prone, redundancy, runtime check.

Tracing - 3

Using Russell's Theory of Types for AOP means:

Tracing - 3

Using Russell's Theory of Types for AOP means:

- An aspect is of higher type level than base code.

Tracing - 3

Using Russell's Theory of Types for AOP means:

- An aspect is of higher type level than base code.
- An aspect advising another aspect
  is of higher level than the advised aspect.

Tracing - 3

Using Russell's Theory of Types for AOP means:

- An aspect is of higher type level than base code.
- An aspect advising another aspect
  is of higher level than the advised aspect.
- Pointcuts in an aspect only select joinpoints
  in the scope of aspects/classes of lower level.

Tracing - 3

Using Russell's Theory of Types for AOP means:

- An aspect is of higher type level than base code.
- An aspect advising another aspect
  is of higher level than the advised aspect.
- Pointcuts in an aspect only select joinpoints
  in the scope of aspects/classes of lower level.
- Syntactical constructs to distinguish levels.

Tracing - 3

Using Russell's Theory of Types for AOP means:

- An aspect is of higher type level than base code.
- An aspect advising another aspect
  is of higher level than the advised aspect.
- Pointcuts in an aspect only select joinpoints
  in the scope of aspects/classes of lower level.
- Syntactical constructs to distinguish levels.

Solution Preview

```
public meta aspect Tracing {
  void around(): adviceexecution()||execution (* *(..)) {
    System.out.println("Entering:" + thisJoinPoint);
    proceed();
    System.out.println("Leaving: " + thisJoinPoint);
  }
}
```

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3		
Level 2		
Level 1		
Level 0	`class...`	

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3		
Level 2		
Level 1		
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3		
Level 2		
Level 1	`aspect...`	
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3		
Level 2		
Level 1	`aspect...`	$<$ `pc` $>$ except `adviceexecution()`
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3		
Level 2	`meta aspect...`	
Level 1	`aspect...`	$<$ `pc` $>$ except `adviceexecution()`
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3		
Level 2	`meta aspect...`	`adviceexecution()`,
Level 1	`aspect...`	$<$ `pc` $>$ except `adviceexecution()`
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3		
Level 2	`meta aspect...`	`adviceexecution()`, `meta` $<$ `pc` $>$
Level 1	`aspect...`	$<$ `pc` $>$ except `adviceexecution()`
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3		
Level 2	`meta aspect...`	`adviceexecution()`, `meta` < `pc` > and < `pc` >
Level 1	`aspect...`	< `pc` > except `adviceexecution()`
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3	`meta`^{2} `aspect...`	
Level 2	`meta aspect...`	`adviceexecution()`, `meta` $<$ `pc` $>$ and $<$ `pc` $>$
Level 1	`aspect...`	$<$ `pc` $>$ except `adviceexecution()`
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3	`meta`^{2} `aspect...`	`meta adviceexecution()`,
Level 2	`meta aspect...`	`adviceexecution()`, `meta` $<$ `pc` $>$ and $<$ `pc` $>$
Level 1	`aspect...`	$<$ `pc` $>$ except `adviceexecution()`
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword meta

Type Level	Type Definition	Allowed Pointcuts
Level 3	meta^2 aspect...	meta adviceexecution(), adviceexecution(),
Level 2	meta aspect...	adviceexecution(), meta $<$ pc $>$ and $<$ pc $>$
Level 1	aspect...	$<$ pc $>$ except adviceexecution()
Level 0	class...	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3	`meta`^{2} `aspect...`	`meta adviceexecution()`, `adviceexecution()`, `meta`^{2} $<$ `pc` $>$,
Level 2	`meta aspect...`	`adviceexecution()`, `meta` $<$ `pc` $>$ and $<$ `pc` $>$
Level 1	`aspect...`	$<$ `pc` $>$ except `adviceexecution()`
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3	$\texttt{meta}^2$ `aspect...`	`meta adviceexecution()`, `adviceexecution()`, $\texttt{meta}^2$ $<$ pc $>$, `meta` $<$ pc $>$
Level 2	`meta aspect...`	`adviceexecution()`, `meta` $<$ pc $>$ and $<$ pc $>$
Level 1	`aspect...`	$<$ pc $>$ except `adviceexecution()`
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
Level 3	`meta`^{2} `aspect...`	`meta adviceexecution()`, `adviceexecution()`, `meta`^{2} $<$ `pc` $>$, `meta` $<$ `pc` $>$ and $<$ `pc` $>$
Level 2	`meta aspect...`	`adviceexecution()`, `meta` $<$ `pc` $>$ and $<$ `pc` $>$
Level 1	`aspect...`	$<$ `pc` $>$ except `adviceexecution()`
Level 0	`class...`	no pointcuts

New Syntax and Semantics

The keyword `meta`

Type Level	Type Definition	Allowed Pointcuts
...	...	...
Level 3	$\texttt{meta}^2$ `aspect`...	`meta adviceexecution()`, `adviceexecution()`, $\texttt{meta}^2$ $<$ pc $>$, `meta` $<$ pc $>$ and $<$ pc $>$
Level 2	`meta aspect`...	`adviceexecution()`, `meta` $<$ pc $>$ and $<$ pc $>$
Level 1	`aspect`...	$<$ pc $>$ except `adviceexecution()`
Level 0	`class`...	no pointcuts

Related Work - 1

Law of Demeter for Aspects [Liebherr, Lorenz, Wu, 2004]

Requirement:
Solution:

Law of Demeter for Aspects [Liebherr, Lorenz, Wu, 2004]

Requirement: Checking the LoD checker

Solution:

Related Work - 1

Law of Demeter for Aspects [Liebherr, Lorenz, Wu, 2004]

Requirement:	Checking the LoD checker
Solution:	`meta aspect LoD`
	`advises aspect LoD`

Related Work - 1

Law of Demeter for Aspects [Liebherr, Lorenz, Wu, 2004]

Requirement: Checking the LoD checker

Solution: `meta aspect LoD`

advises `aspect LoD`

Testing Aspects [Sokenou, Hermann, 2005]

Requirement:

Solution:

Related Work - 1

Law of Demeter for Aspects [Liebherr, Lorenz, Wu, 2004]

Requirement: Checking the LoD checker

Solution: `meta aspect LoD`
`advises aspect LoD`

Testing Aspects [Sokenou, Hermann, 2005]

Requirement: "We need join points for advices to instrument aspects as well"

Solution:

Related Work - 1

Law of Demeter for Aspects [Liebherr, Lorenz, Wu, 2004]

Requirement:	Checking the LoD checker
Solution:	`meta aspect LoD` `advises aspect LoD`

Testing Aspects [Sokenou, Hermann, 2005]

Requirement:	"We need join points for advices to instrument aspects as well"
Solution:	`meta aspect AspectTester` `instruments aspect ObjectAdvisor`

Conclusion and Outlook

Results

- Restoration of intuitive semantics (no more paradoxes!)

Conclusion and Outlook

Results

- Restoration of intuitive semantics (no more paradoxes!)
- Clear seperation of aspects.

Conclusion and Outlook

Results

- Restoration of intuitive semantics (no more paradoxes!)
- Clear seperation of aspects.
- Type system prevents programming errors.

Conclusion and Outlook

Results

- Restoration of intuitive semantics (no more paradoxes!)
- Clear seperation of aspects.
- Type system prevents programming errors.
- (Slightly) more expressive pointcut language.

Conclusion and Outlook

Results

- Restoration of intuitive semantics (no more paradoxes!)
- Clear seperation of aspects.
- Type system prevents programming errors.
- (Slightly) more expressive pointcut language.

Further Work

- Implementation of the type system (with abc group).

Conclusion and Outlook

Results

- Restoration of intuitive semantics (no more paradoxes!)
- Clear seperation of aspects.
- Type system prevents programming errors.
- (Slightly) more expressive pointcut language.

Further Work

- Implementation of the type system (with abc group).
- Systematic migration of current AOP applications.

Conclusion and Outlook

Results

- Restoration of intuitive semantics (no more paradoxes!)
- Clear seperation of aspects.
- Type system prevents programming errors.
- (Slightly) more expressive pointcut language.

Further Work

- Implementation of the type system (with abc group).
- Systematic migration of current AOP applications.
- Collect empirical data on how many levels and/or whether intermediate levels are needed.

Conclusion and Outlook

Results

- Restoration of intuitive semantics (no more paradoxes!)
- Clear seperation of aspects.
- Type system prevents programming errors.
- (Slightly) more expressive pointcut language.

Further Work

- Implementation of the type system (with abc group).
- Systematic migration of current AOP applications.
- Collect empirical data on how many levels and/or whether intermediate levels are needed.
- Generalization of the type system.