StrongAspectJ

Flexible and Safe Pointcut/Advice Bindings

Bruno De Fraine[1], Mario Südholt[2] and Viviane Jonckers[1]

[1]System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussel

[2]OBASCO project
École des Mines de Nantes-INRIA, LINA

April 2, 2008 at AOSD'08

Problem Context

- AspectJ, CaesarJ, JAsCo,...
  - Seamless AOP extensions of Java
  - Typed, but safe and expressive?

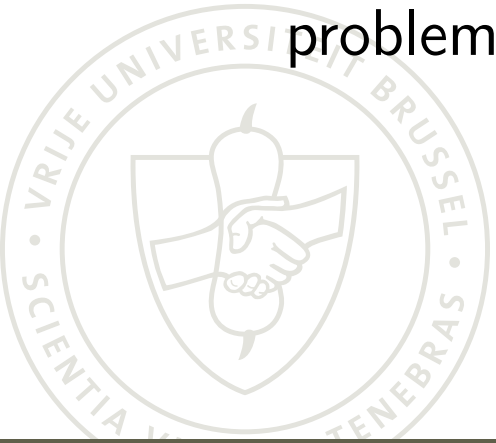
Problem Context

- AspectJ, CaesarJ, JAsCo,...
  - Seamless AOP extensions of Java
  - Typed, but safe and expressive?
- Current aspect typing falls short

  Safety: uncaught type errors

  Expressiveness: valid, useful aspects are restricted

  Previous work only partially identified and/or solved these problems



Problem Context

- AspectJ, CaesarJ, JAsCo,...
  - Seamless AOP extensions of Java
  - Typed, but safe and expressive?
- Current aspect typing falls short

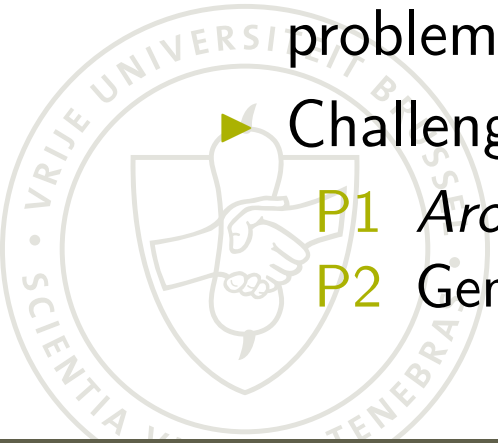
  Safety: uncaught type errors

  Expressiveness: valid, useful aspects are restricted

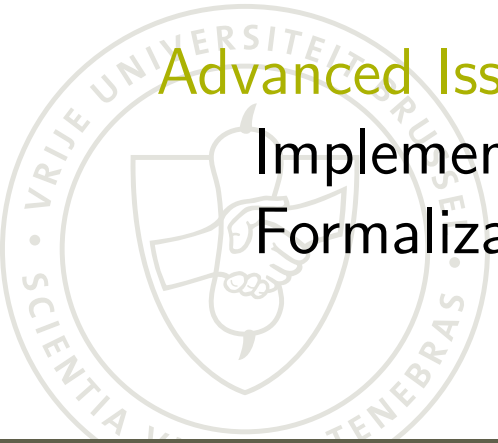
  Previous work only partially identified and/or solved these problems
- Challenges of typing pointcuts and advice

  P1 *Around*-advice and *proceed* complicate subtype variance

  P2 Generic advice applies to heterogeneous join points



Outline



AspectJ Typing: Subtype Variance Example

```
pointcut pc(Employee e): args(e) && call(Number *(..));
Integer around(Person p): pc(p) { ... }
```

- Advice types differ from pointcut types
  - Provides more (result covariant)
  - Expects less (args contravariant)

$$\texttt{Integer} <: \texttt{Number}$$
$$\texttt{Employee} <: \texttt{Person}$$

AspectJ Typing: Subtype Variance Example

```
pointcut pc(Employee e): args(e) && call(Number *(..));
Integer around(Person p): pc(p) { ... }
```

- Advice types differ from pointcut types
  - Provides more (result covariant)
  - Expects less (args contravariant)

$$\texttt{Integer} <: \texttt{Number}$$
$$\texttt{Employee} <: \texttt{Person}$$

AspectJ Typing: Subtype Variance Example

```
pointcut pc(Employee e): args(e) && call(Number *(..));
Integer around(Person p): pc(p) { ... }
```

- Advice types differ from pointcut types
  - Provides more (result covariant)
  - Expects less (args contravariant)

$$\texttt{Integer} <: \texttt{Number}$$
$$\texttt{Employee} <: \texttt{Person}$$

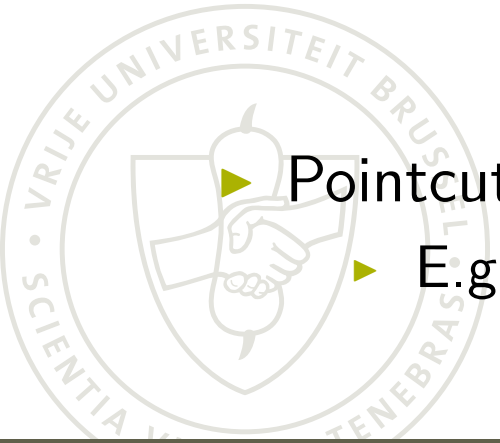
AspectJ Typing: Subtype Variance Example

```
pointcut pc(Employee e): args(e) && call(Number *(..));
Integer around(Person p): pc(p) { ... }
```

- Advice types differ from pointcut types
  - Provides more (result covariant)
  - Expects less (args contravariant)

$$\texttt{Integer} <: \texttt{Number}$$
$$\texttt{Employee} <: \texttt{Person}$$

- Pointcut types may further differ from join point types
  - E.g. `args` matches subtype instances as well



AspectJ Typing: General Rules

- Substitution principle: method σ can be executed in place of method σ', if $\sigma \leq \sigma'$, where

$$R(S_1, S_2, \ldots) \leq R'(S'_1, S'_2, \ldots) \text{ iff } R <: R' \text{ and } S'_i <: S_i$$

AspectJ Typing: General Rules

- Substitution principle: method σ can be executed in place of method σ', if $\sigma \leq \sigma'$, where

  $$R(S_1, S_2, \ldots) \leq R'(S'_1, S'_2, \ldots) \text{ iff } R <: R' \text{ and } S'_i <: S_i$$

- Type relations:

Matching	$\sigma_{pointcut} \leq \sigma_{join\ point}$
Binding	$\sigma_{advice} \leq \sigma_{pointcut}$

AspectJ Typing: General Rules

- Substitution principle: method σ can be executed in place of method σ', if $\sigma \leq \sigma'$, where

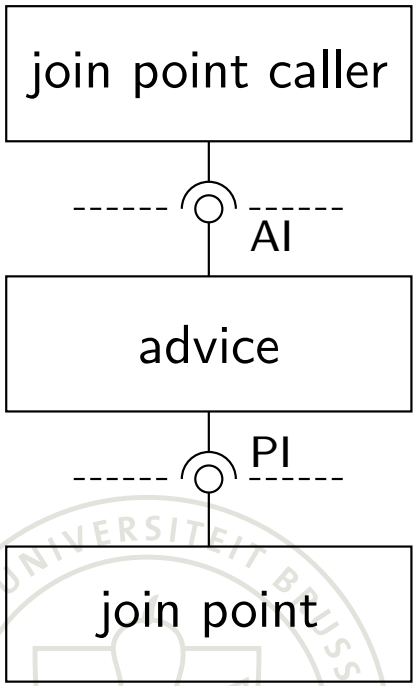
  $$R(S_1, S_2, \ldots) \leq R'(S'_1, S'_2, \ldots) \text{ iff } R <: R' \text{ and } S'_i <: S_i$$

- Type relations:

Matching	$\sigma_{pointcut} \leq \sigma_{join\ point}$
Binding	$\sigma_{advice} \leq \sigma_{pointcut}$

- Advice can safely replace or augment join point

P1: Taking into Account *proceed*

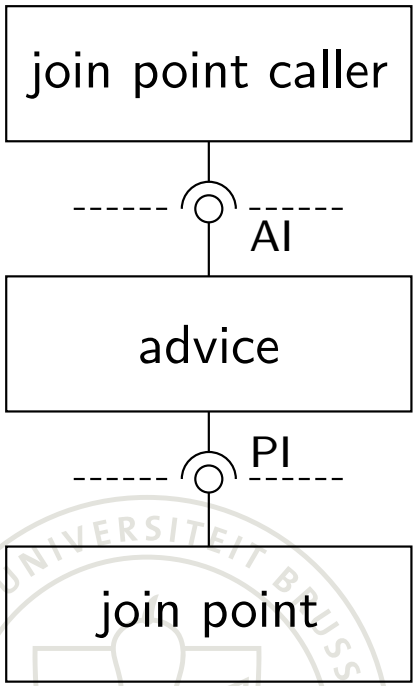


- *Around*-advice: two interfaces

  Advice interface *provided* to join point caller

  Proceed interface *expected* from join point

P1: Taking into Account *proceed*



- *Around*-advice: two interfaces

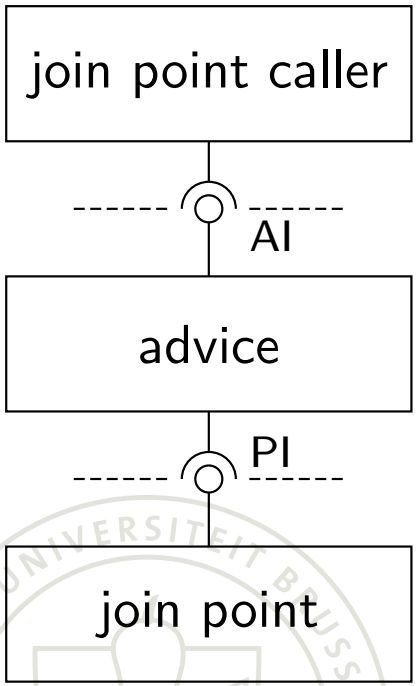
  Advice interface *provided* to join point caller

  Proceed interface *expected* from join point
- AspectJ uses only one signature

$$\sigma_{proceed} = \sigma_{advice}$$

I.e. advice can expect whatever it provides

P1: Taking into Account *proceed*



- *Around*-advice: two interfaces

  Advice interface *provided* to join point caller
  Proceed interface *expected* from join point
- AspectJ uses only one signature

$$\sigma_{proceed} = \sigma_{advice}$$

  I.e. advice can expect whatever it provides
- Incorrect assumption in case of type variance!
  - Advice provides more than the join point it replaces

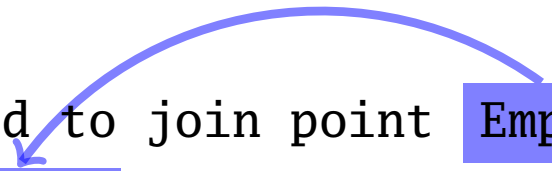
Loophole: *Proceed*-Method

- Advice accepts any `Person`

```
// Will be applied to join point  Employee.promote()
void around( Person p ): this(p) {
    proceed( new Person() );
}
```

Loophole: *Proceed*-Method

- Advice accepts any `Person`



```
// Will be applied to join point Employee.promote()
void around( Person p ): this(p) {
    proceed( new Person() );
}
```

Loophole: *Proceed*-Method

- Advice accepts any `Person`
  - Join point might not: `ClassCastException`

```
// Will be applied to join point Employee.promote()
void around( Person p ): this(p) {
    proceed( new Person() );
}
```

Loophole: *Proceed*-Method

- Advice accepts any `Person`
  - Join point might not: `ClassCastException`

```
// Will be applied to join point  Employee.promote()
void around( Person p ): this(p) {
    proceed( new Person() );
}
```

- Analogous loophole when returning values from proceed

Loophole: *Proceed*-Method

- Advice accepts any `Person`
  - Join point might not: `ClassCastException`

```
// Will be applied to join point  Employee.promote()
void around( Person p ): this(p) {
    proceed( new Person() );
}
```

- Analogous loophole when returning values from proceed
- [Wand et al., TOPLAS04] identifies the loopholes
- [Clifton and Leavens, SCP06]: modified, safe proceed

P2: Generic Advice (`Object` return type) I

- Not every *around*-advice employs full replacement power
  - Very common not to do so, e.g. profiling advice

P2: Generic Advice (`Object` return type) I

- Not every *around*-advice employs full replacement power
  - Very common not to do so, e.g. profiling advice
- Generic advice: compatible with a larger set of join points
  1. Always *proceed* with original argument
  2. Always return value obtained from *proceed*

  ($\neq$ support for base language generics as in [AspectJ 5])

P2: Generic Advice (`Object` return type) I

- Not every *around*-advice employs full replacement power
  - Very common not to do so, e.g. profiling advice
- Generic advice: compatible with a larger set of join points
  1. Always *proceed* with original argument
  2. Always return value obtained from *proceed*

  ($\neq$ support for base language generics as in [AspectJ 5])
- Ad hoc AspectJ type rule: `Object` return type always valid

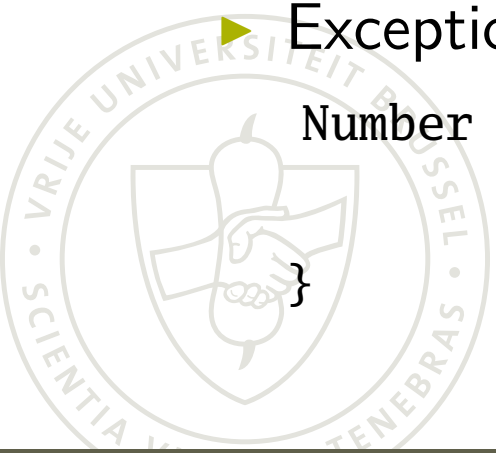
P2: Generic Advice (`Object` return type) I

- Not every *around*-advice employs full replacement power
  - Very common not to do so, e.g. profiling advice
- Generic advice: compatible with a larger set of join points
  1. Always *proceed* with original argument
  2. Always return value obtained from *proceed*

  ($\neq$ support for base language generics as in [AspectJ 5])
- Ad hoc AspectJ type rule: `Object` return type always valid
- Exception to variance rule:

```
Number around(): call( Integer *(..)) {

}
```



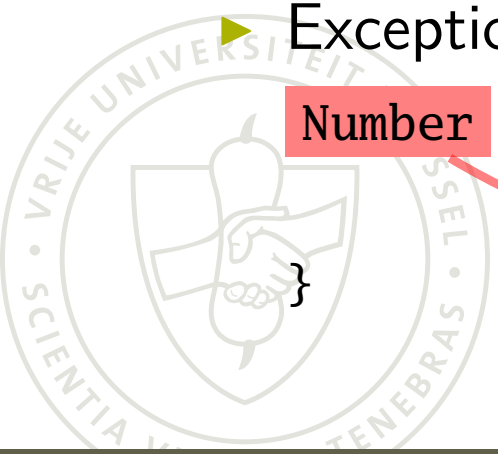
P2: Generic Advice (`Object` return type) I

- Not every *around*-advice employs full replacement power
  - Very common not to do so, e.g. profiling advice
- Generic advice: compatible with a larger set of join points
  1. Always *proceed* with original argument
  2. Always return value obtained from *proceed*

  ($\neq$ support for base language generics as in [AspectJ 5])
- Ad hoc AspectJ type rule: `Object` return type always valid
- Exception to variance rule:

```
Number around(): call( Integer *(..)) {
}
```

rejected



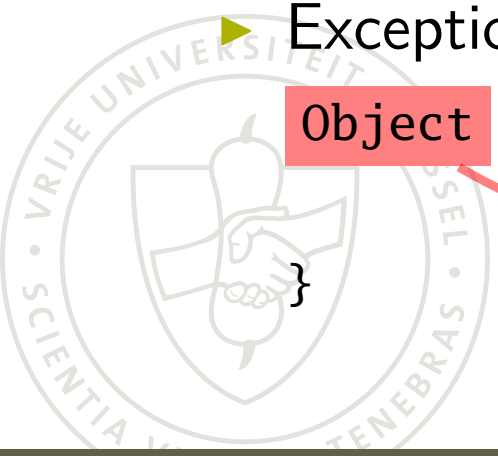
P2: Generic Advice (`Object` return type) I

- Not every *around*-advice employs full replacement power
  - Very common not to do so, e.g. profiling advice
- Generic advice: compatible with a larger set of join points
  1. Always *proceed* with original argument
  2. Always return value obtained from *proceed*

  ($\neq$ support for base language generics as in [AspectJ 5])
- Ad hoc AspectJ type rule: `Object` return type always valid
- Exception to variance rule:

```
Object around(): call(Integer *(..)) {
}
```

accepted!

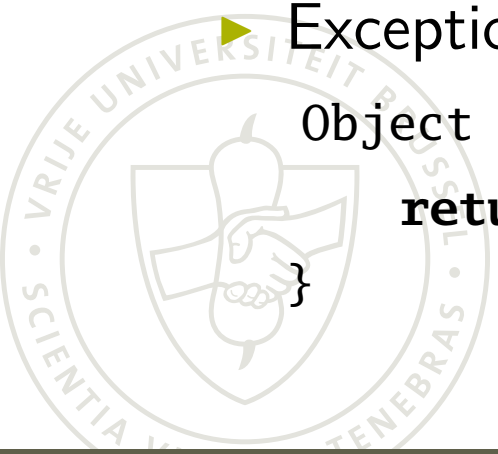


P2: Generic Advice (`Object` return type) I

- Not every *around*-advice employs full replacement power
  - Very common not to do so, e.g. profiling advice
- Generic advice: compatible with a larger set of join points
  1. Always *proceed* with original argument
  2. Always return value obtained from *proceed*

  ($\neq$ support for base language generics as in [AspectJ 5])
- Ad hoc AspectJ type rule: `Object` return type always valid
- Exception to variance rule: not safe!

```
Object around(): call( Integer *(..)) {
    return new Object();
}
```



Generic Advice (`Object` return type) II

- Safe but restricted advice:

```
Number around(): call(( Integer || Float ) *(..)) {
    Number n = proceed() ;
    while(n.intValue() > 100)
        n = proceed() ;
    return n ;
}
```

Generic Advice (`Object` return type) II

- Safe but restricted advice:

```
Number around(): call(( Integer || Float ) *(..)) {
    Number n = proceed();
    while(n.intValue() > 100)
        n = proceed();
    return n;
}
```

Generic Advice (`Object` return type) II

- Safe but restricted advice:

```
Number around(): call(( Integer || Float ) *(..)) {
    Number n = proceed() ;
    while(n.intValue() > 100)
        n = proceed() ;
    return n ;
}
```

Generic Advice (`Object` return type) II

- Safe but restricted advice:

```
Number around(): call(( Integer || Float ) *(..)) {
    Number n = proceed() ;
    while(n.intValue() > 100)
        n = proceed() ;
    return n ;
}
```

- [Jagadeesan et al., SCP06, Masuhara et al., ICFP05]
  - Identify loophole
  - Propose typing generic advice using type variables

StrongAspectJ: Overview

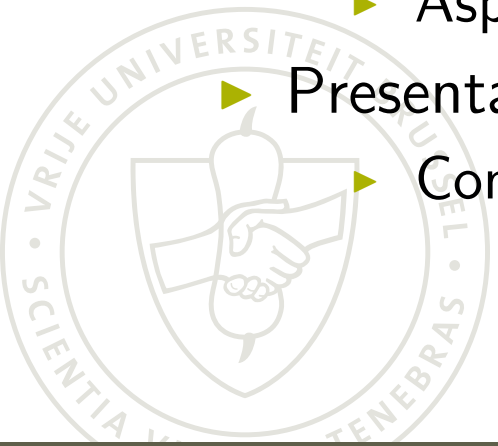
- Novel type system designed to tackle these problems
  - Support generic/non-generic advice behavior (with proceed)
  - Recover safety, extend previous results

StrongAspectJ: Overview

- Novel type system designed to tackle these problems
  - Support generic/non-generic advice behavior (with proceed)
  - Recover safety, extend previous results
- Specified on two levels
  - General typing principles
  - AspectJ integration: StrongAspectJ language

StrongAspectJ: Overview

- Novel type system designed to tackle these problems
  - Support generic/non-generic advice behavior (with proceed)
  - Recover safety, extend previous results
- Specified on two levels
  - General typing principles
  - AspectJ integration: StrongAspectJ language
- Presentation: principles with StrongAspectJ examples
  - Complete language definition in paper



Typing Principle I: Dual Advice Signature

σ_{advice} and $\sigma_{proceed}$ may be different

```
void around(Object o): ....: void proceed(Employee) {

}
```

Typing Principle I: Dual Advice Signature

σ_{advice} and $\sigma_{proceed}$ may be different

```
void around(Object o): ....: void proceed(Employee) {
    proceed(new Employee()); //OK

}
```

Typing Principle I: Dual Advice Signature

σ_{advice} and $\sigma_{proceed}$ may be different

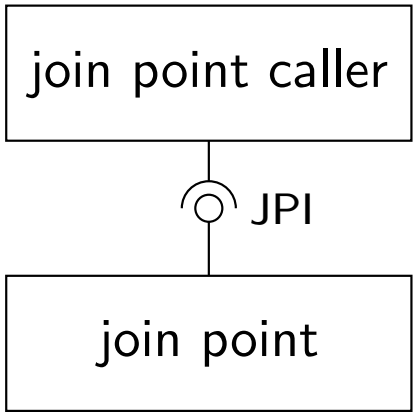
```
void around(Object o): ....: void proceed(Employee) {
    proceed(new Employee()); //OK
    proceed(o);              //Err!
}
```

Typing Principle II: Variance Relations

Required relations between σ_{advice}, $\sigma_{proceed}$ and $\sigma_{join\ point}$?

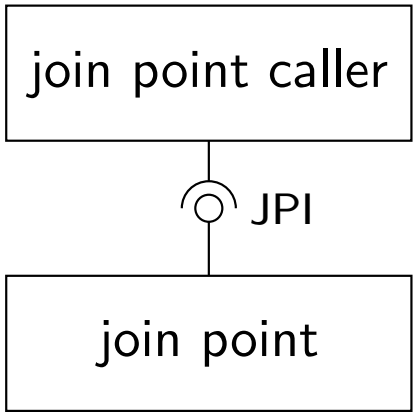
Typing Principle II: Variance Relations

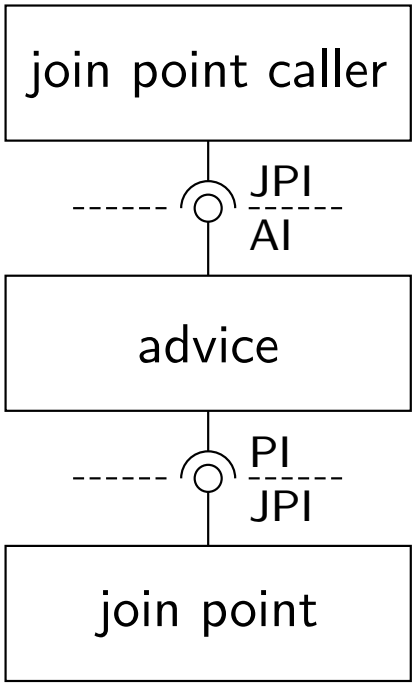
Required relations between σ_{advice}, $\sigma_{proceed}$ and $\sigma_{join\ point}$?



Typing Principle II: Variance Relations

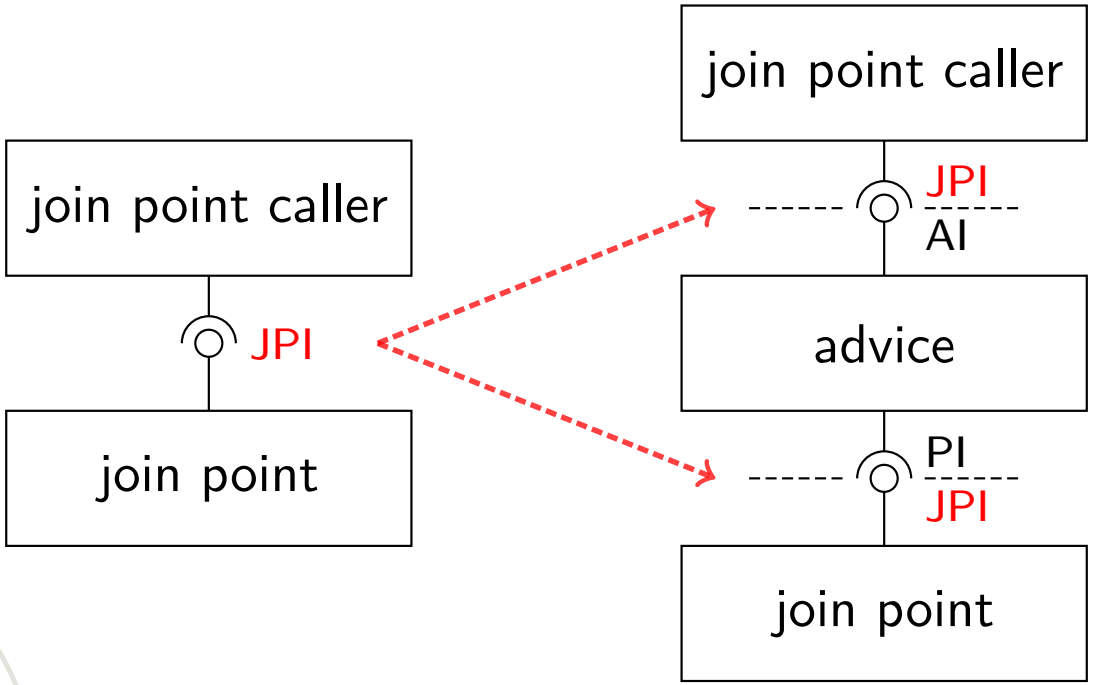
Required relations between σ_{advice}, $\sigma_{proceed}$ and $\sigma_{join\ point}$?





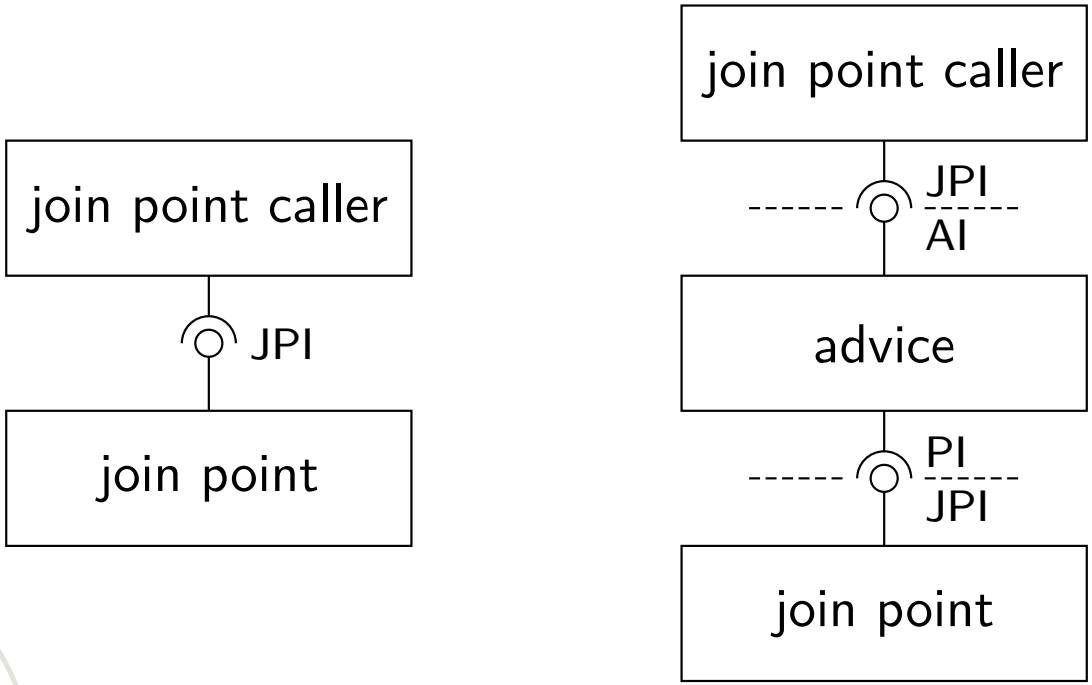
Typing Principle II: Variance Relations

Required relations between σ_{advice}, $\sigma_{proceed}$ and $\sigma_{join\ point}$?



Typing Principle II: Variance Relations

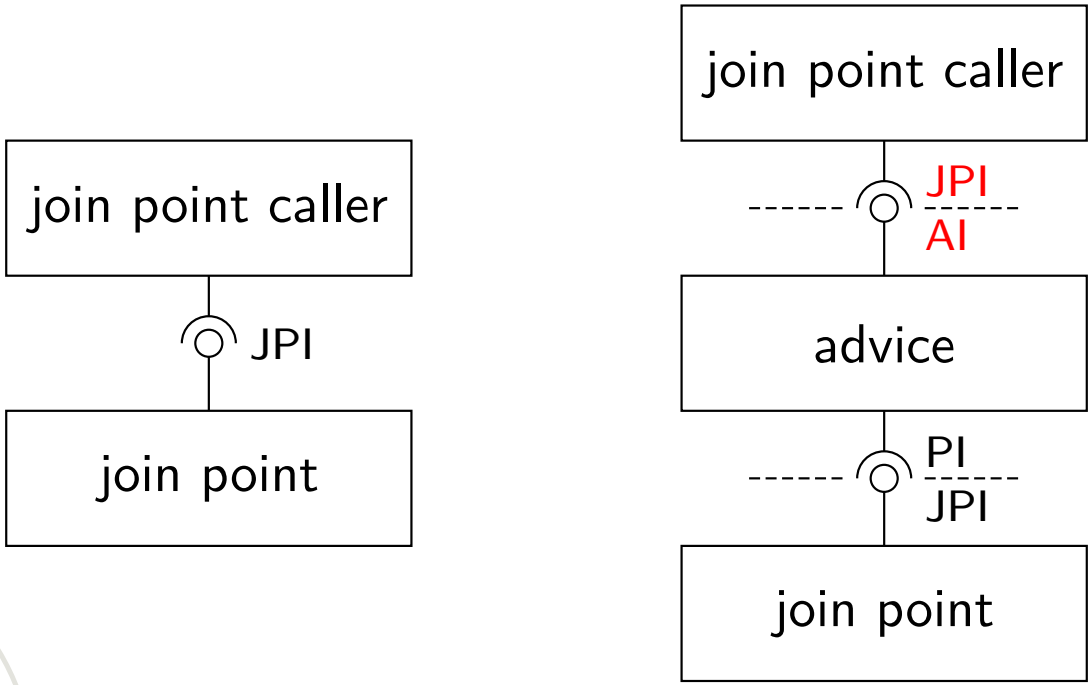
Required relations between σ_{advice}, $\sigma_{proceed}$ and $\sigma_{join\ point}$?



$$\sigma_{advice} \leq \sigma_{join\ point} \leq \sigma_{proceed}$$

Typing Principle II: Variance Relations

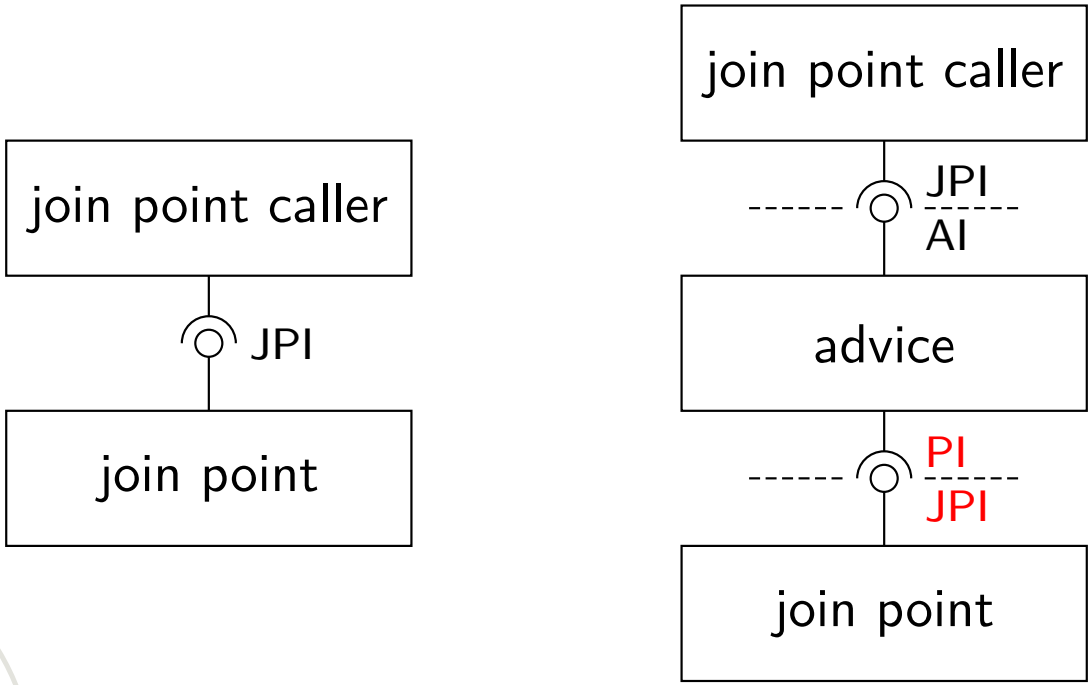
Required relations between σ_{advice}, $\sigma_{proceed}$ and $\sigma_{join\ point}$?



$$\sigma_{advice} \leq \sigma_{join\ point} \leq \sigma_{proceed}$$

Typing Principle II: Variance Relations

Required relations between σ_{advice}, $\sigma_{proceed}$ and $\sigma_{join\ point}$?



$$\sigma_{advice} \leq \sigma_{join\ point} \leq \sigma_{proceed}$$

Typing Principle III: Pointcuts and Type Variables

- Pointcut signature range: lower bound - upper bound

Typing Principle III: Pointcuts and Type Variables

- Pointcut signature range: lower bound - upper bound

- Example:

```
pointcut pc( Employee – Person  p): this(p);
```

Typing Principle III: Pointcuts and Type Variables

- Pointcut signature range: lower bound - upper bound

  Matching $\sigma_{pc,lower} \leq \sigma_{join\ point} \leq \sigma_{pc,upper}$

- Example:

```
pointcut pc( Employee - Person  p): this(p);
// Matches Employee.promote(), Person.getAge(),...
// Does not match Manager.report(..)
```

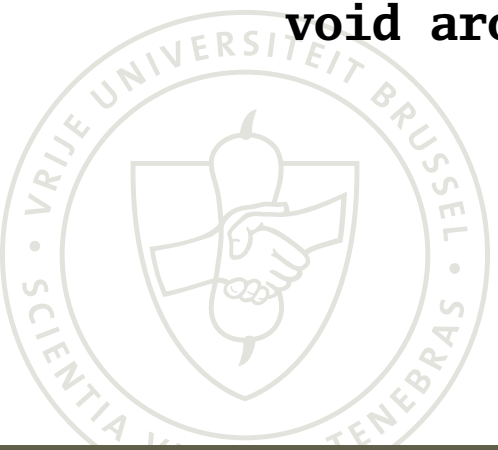
Typing Principle III: Pointcuts and Type Variables

- Pointcut signature range: lower bound - upper bound

Matching	$\sigma_{pc,lower} \leq \sigma_{join\ point} \leq \sigma_{pc,upper}$	
Binding	$\sigma_{advice} \leq \sigma_{pc,lower}$	$\sigma_{pc,upper} \leq \sigma_{proceed}$

- Example:

```
pointcut pc( Employee - Person  p): this(p);
// Matches Employee.promote(), Person.getAge(),...
// Does not match Manager.report(..)
void around( Object  o): pc(o): proceed( Employee ) { }
```



Typing Principle III: Pointcuts and Type Variables

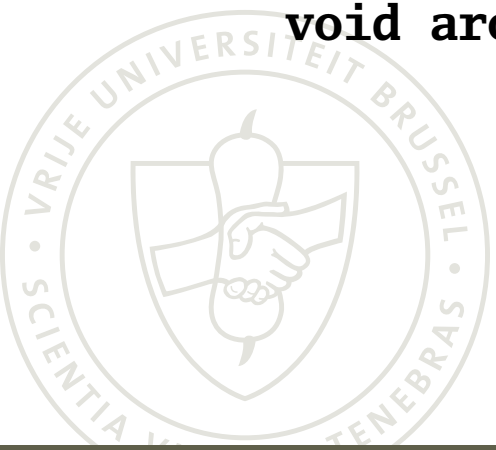
- Pointcut signature range: lower bound - upper bound

Matching $\sigma_{pc,lower} \leq \sigma_{join\ point} \leq \sigma_{pc,upper}$

Binding $\sigma_{advice} \leq \sigma_{pc,lower}$ $\sigma_{pc,upper} \leq \sigma_{proceed}$

- Example:

```
pointcut pc( Employee – Person p): this(p);
// Matches Employee.promote(), Person.getAge(),...
// Does not match Manager.report(..)
void around( Object o): pc(o): proceed( Employee ) { }
```



Typing Principle III: Pointcuts and Type Variables

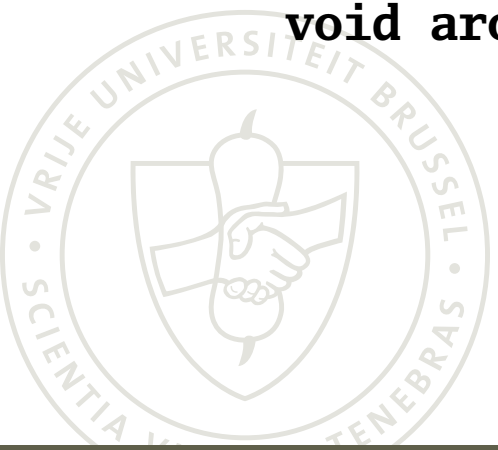
- Pointcut signature range: lower bound - upper bound

Matching $\sigma_{pc,lower} \leq \sigma_{join\ point} \leq \sigma_{pc,upper}$

Binding $\sigma_{advice} \leq \sigma_{pc,lower}$ $\sigma_{pc,upper} \leq \sigma_{proceed}$

- Example:

```
pointcut pc( Employee - Person  p): this(p);
// Matches Employee.promote(), Person.getAge(),...
// Does not match Manager.report(..)
void around( Object  o): pc(o): proceed( Employee ) { }
```



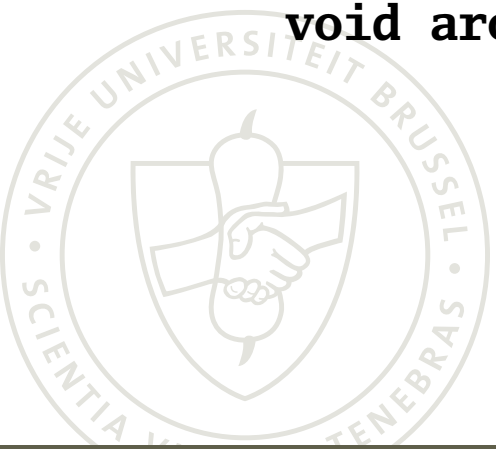
Typing Principle III: Pointcuts and Type Variables

- Pointcut signature range: lower bound - upper bound

Matching	$\sigma_{pc,lower} \leq \sigma_{join\ point} \leq \sigma_{pc,upper}$	
Binding	$\sigma_{advice} \leq \sigma_{pc,lower}$	$\sigma_{pc,upper} \leq \sigma_{proceed}$

- Example:

```
pointcut pc( Employee - Person  p): this(p);
// Matches Employee.promote(), Person.getAge(),...
// Does not match Manager.report(..)
void around( Object  o): pc(o): proceed( Employee ) { }
```



Typing Principle III: Pointcuts and Type Variables

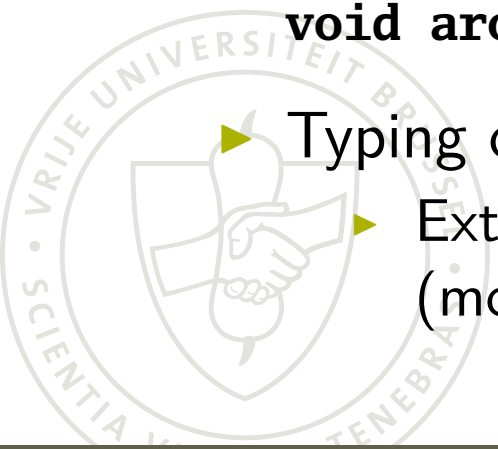
- Pointcut signature range: lower bound - upper bound

Matching	$\sigma_{pc,lower} \leq \sigma_{join\ point} \leq \sigma_{pc,upper}$	
Binding	$\sigma_{advice} \leq \sigma_{pc,lower}$	$\sigma_{pc,upper} \leq \sigma_{proceed}$

- Example:

```
pointcut pc( Employee - Person  p): this(p);
// Matches Employee.promote(), Person.getAge(),...
// Does not match Manager.report(..)
void around( Object  o): pc(o): proceed( Employee ) { }
```

- Typing of generic advices using type variables
  - Extension of type relations to include type variables (more details in paper)



Practical Example: Factory Method Specialization

```
class Factory {
  Component createTextArea(String t) { ... }
}
aspect ScrollPaneFactory {
  pointcut creation(): execution(* Factory.create*(..));
  JScrollPane around(): creation(): Component proceed() {
    return new JScrollPane(proceed());
  }
}
```

- Advice wraps created component in scroll pane

Practical Example: Factory Method Specialization

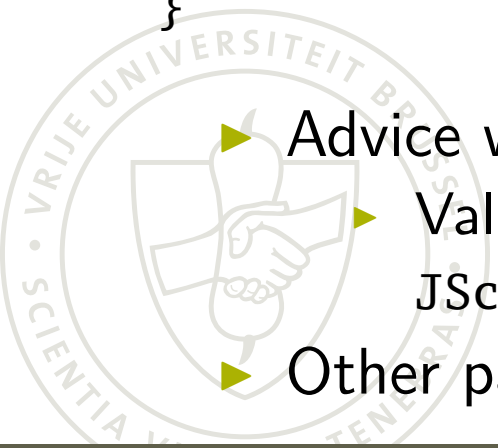
```
class Factory {
  Component createTextArea(String t) { ... }
}
aspect ScrollPaneFactory {
  pointcut creation(): execution(* Factory.create*(..));
  JScrollPane around(): creation(): Component proceed() {
    return new JScrollPane(proceed());
  }
}
```

- Advice wraps created component in scroll pane
  - Valid for join points within range
    `JScrollPane` – `Component`

Practical Example: Factory Method Specialization

```
class Factory {
  Component createTextArea(String t) { ... }
}
aspect ScrollPaneFactory {
  pointcut creation(): execution(* Factory.create*(..));
  JScrollPane around(): creation(): Component proceed() {
    return new JScrollPane(proceed());
  }
}
```

- Advice wraps created component in scroll pane
  - Valid for join points within range
    `JScrollPane – Component`
- Other paper examples: Caching, Profiling

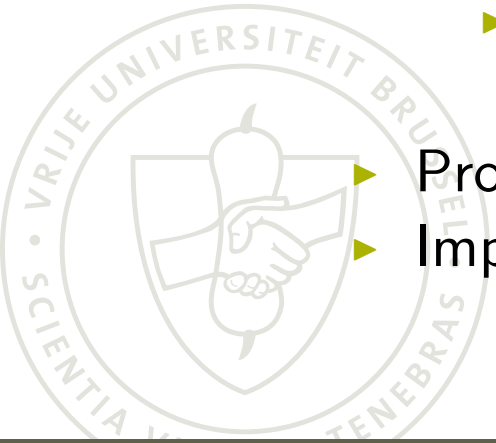


Two Implementations

1. StrongAJ: plugin for AspectBench Compiler (abc) 1.2.1
   - Implements complete StrongAspectJ proposal
   - Test suite of static and dynamic tests
   - Source code available at
     `http://ssel.vub.ac.be/strongaj/`

Two Implementations

1. StrongAJ: plugin for AspectBench Compiler (abc) 1.2.1
   - Implements complete StrongAspectJ proposal
   - Test suite of static and dynamic tests
   - Source code available at
     `http://ssel.vub.ac.be/strongaj/`
2. Type rules encoded in interfaces of AOP framework
   - Plain Java 5: type variables, wildcard type arguments
     - Type restrictions enforced using unmodified generics-aware compiler
   - Proof-of-concept in paper
   - Improves AOP Alliance, AspectJ annotation-style,...



Formalization

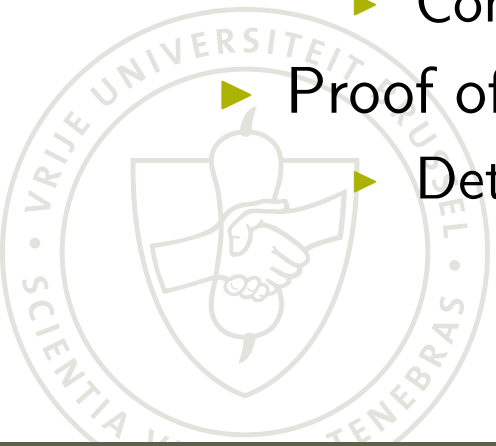
- Formal definition of evaluation and typing rules
  - Extension of [Jagadeesan et al., SCP06] framework (Featherweight Java calculus)

Formalization

- Formal definition of evaluation and typing rules
  - Extension of [Jagadeesan et al., SCP06] framework (Featherweight Java calculus)
- Pointcut language is abstract
  - No definition of pointcut matching and typing judgements, only form
  - Consistency property: matched join points satisfy typing

Formalization

- Formal definition of evaluation and typing rules
  - Extension of [Jagadeesan et al., SCP06] framework (Featherweight Java calculus)
- Pointcut language is abstract
  - No definition of pointcut matching and typing judgements, only form
  - Consistency property: matched join points satisfy typing
- Proof of type safety properties: preservation and progress
  - Details in paper and companion technical report



Conclusions

- Novel type system for pointcut and advice mechanism

Conclusions

- Novel type system for pointcut and advice mechanism
  - Main elements:
    - Dual advice signature
    - Pointcut type ranges

Conclusions

- Novel type system for pointcut and advice mechanism
  - Main elements:
    - Dual advice signature
    - Pointcut type ranges
  - Integrates support for generic advice

Conclusions

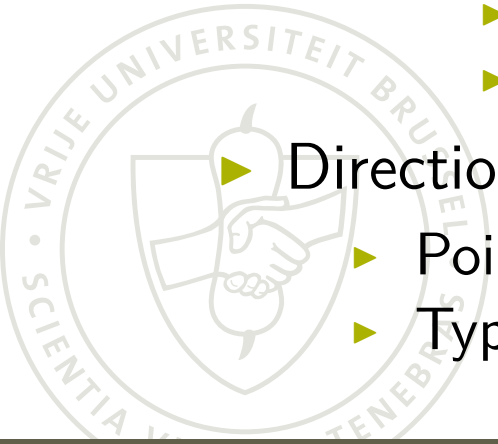
- Novel type system for pointcut and advice mechanism
  - Main elements:
    - Dual advice signature
    - Pointcut type ranges
  - Integrates support for generic advice
  - Avoids current safety loopholes
    - Formal proof of type safety properties

Conclusions

- Novel type system for pointcut and advice mechanism
  - Main elements:
    - Dual advice signature
    - Pointcut type ranges
  - Integrates support for generic advice
  - Avoids current safety loopholes
    - Formal proof of type safety properties
  - Concrete integration in mainstream approaches
    - StrongAspectJ, StrongAJ plugin for abc
    - AOP framework interfaces using Java 5 Generics

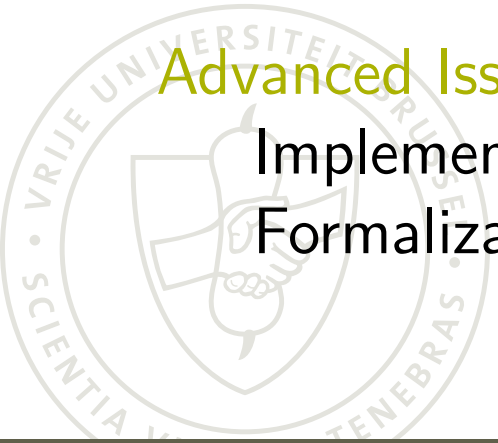
Conclusions

- Novel type system for pointcut and advice mechanism
  - Main elements:
    - Dual advice signature
    - Pointcut type ranges
  - Integrates support for generic advice
  - Avoids current safety loopholes
    - Formal proof of type safety properties
  - Concrete integration in mainstream approaches
    - StrongAspectJ, StrongAJ plugin for abc
    - AOP framework interfaces using Java 5 Generics
- Directions for future research
  - Pointcut typing beyond arguments and result type
  - Type inference of aspect declarations



Discussion

- Motivation
  - AspectJ Typing Rules
  - Loopholes and Restrictions
- StrongAspectJ Proposal
  - Overview
  - Illustrated Typing Principles
- Advanced Issues
  - Implementations
  - Formalization



Selected Related Work I


C. Clifton and G. T. Leavens.
MiniMAO1: An imperative core language for studying aspect-oriented reasoning.
Science of Computer Programming, 63(3):321–374, 2006.

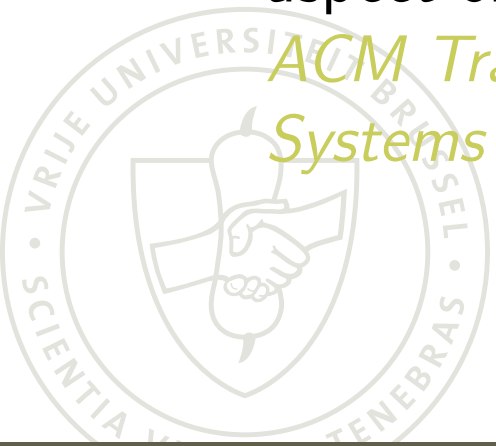

A. Colyer et al.
The AspectJ 5 development kit developer's notebook.
Available at `http://www.eclipse.org/aspectj/doc/released/adk15notebook/`, Dec. 2005.


R. Jagadeesan, A. Jeffrey, and J. Riely.
Typed parametric polymorphism for aspects.
Science of Computer Programming, 63(3):267–296, 2006.

Selected Related Work II


H. Masuhara, H. Tatsuzawa, and A. Yonezawa.
Aspectual Caml: an aspect-oriented functional language.
In *Proc. of ICFP '05*, pages 320–330. ACM Press, 2005.


M. Wand, G. Kiczales, and C. Dutchyn.
A semantics for advice and dynamic join points in aspect-oriented programming.
ACM Transactions on Programming Languages and Systems (TOPLAS), 26(5):890–910, 2004.



Formalization: Consistency Property

$$\textsc{Consistency-PC}\quad \frac{\begin{array}{c}\vdash \mathsf{match}(M,\phi) = \bar{y};\bar{K};M' \\ \vdash \phi : D\text{–}E(\bar{F}\text{–}\bar{G}\ \bar{z}) \qquad \vdash M : R\end{array}}{\begin{array}{c}\bar{y} = \bar{z} \qquad \vdash \bar{K} : \bar{G}' \qquad \vdash \bar{G}' <: \bar{G} \qquad \bullet;\bar{F}\ \bar{y} \vdash M' : R' \\ \vdash R' <: E \qquad \vdash D <: R\end{array}}$$

Formalization: Advice Typing

DEC-ADVICE

$$\frac{\begin{array}{c}\bar{X} \triangleleft \bar{C} \vdash \bar{C}, R, \bar{P}, S, \bar{Q} \\ \bar{X} \triangleleft \bar{C}; \bar{P}\ \bar{x}, S\ \texttt{proceed}(\bar{Q}) \vdash M : R' \\ \bar{X} \triangleleft \bar{C} \vdash R' <: R \\ \vdash \phi : D\text{–}E(\bar{F}\text{–}\bar{G}\ \bar{z}) \\ \bar{Z}, W \text{ fresh} \\ \exists \bar{V}. \left(\begin{array}{cc} \bar{X} \triangleleft \bar{C}, \bar{Z} \triangleleft \bar{G}, W \triangleleft E \vdash \bar{V}, \bar{V} <: \bar{C} & \\ \bar{Z} \triangleleft \bar{G} \vdash \bar{x}[\bar{Z}/\bar{z}] <: \bar{P}[\bar{V}/\bar{x}] & \bar{Z} \triangleright \bar{F} \vdash \bar{Q}[\bar{V}/\bar{x}] <: \bar{x}[\bar{Z}/\bar{z}] \\ W \triangleleft E \vdash W <: S[\bar{V}/\bar{x}] & W \triangleright D \vdash R[\bar{V}/\bar{x}] <: W \end{array} \right) \end{array}}{\vdash \texttt{advice}\ a\texttt{<}\bar{X} \triangleleft \bar{C}\texttt{>}R(\bar{P}\ \bar{x})\texttt{:}\phi\texttt{:}S(\bar{Q})\{M\}}$$