Taming Aspects

Éric Tanter
University of Chile

Aspects?

What are Aspects?

Modular implementation of **crosscutting** concerns

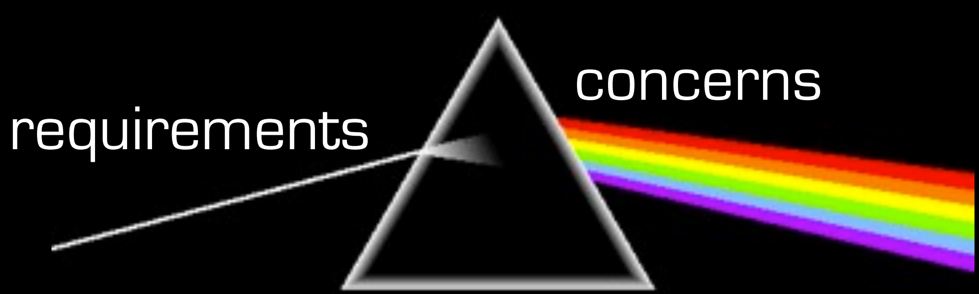
What are Aspects?

Modular implementation of **crosscutting** concerns

What are Aspects?

Modular implementation of **crosscutting** concerns

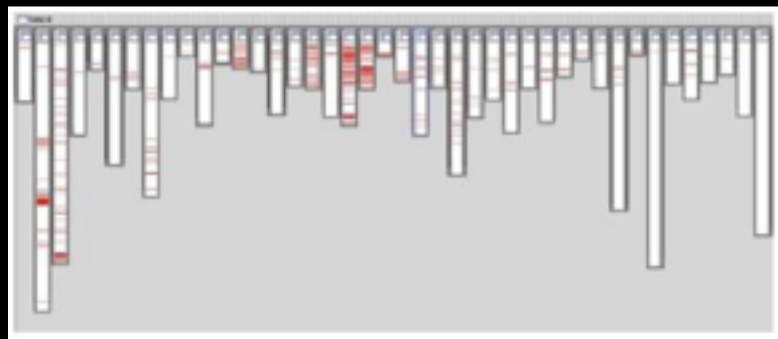
Monitoring
Security
Coordination
...



What are Aspects?

Modular implementation of **crosscutting** concerns

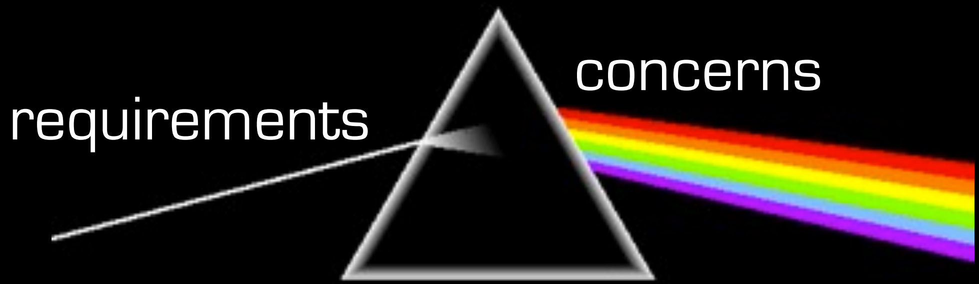
Monitoring
Security
Coordination
...



What are Aspects?

Modular implementation of **crosscutting** concerns

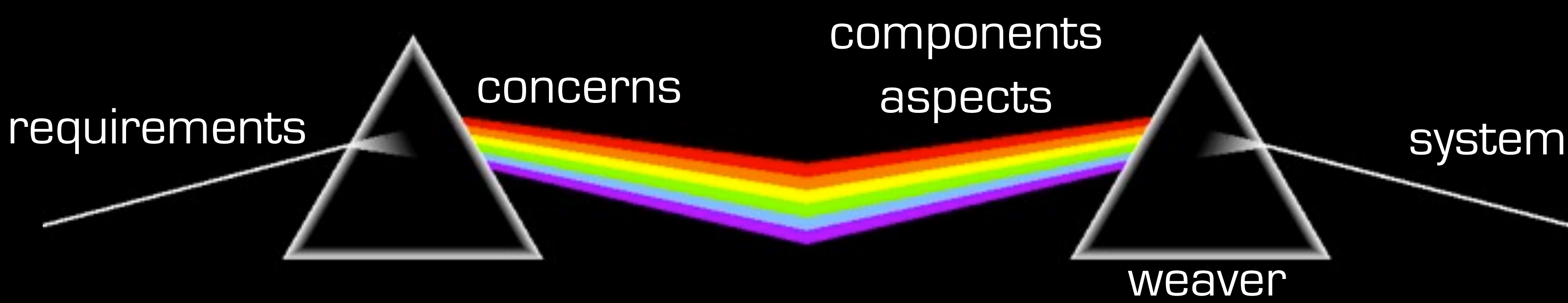
Monitoring
Security
Coordination
...



What are Aspects?

Modular implementation of **crosscutting** concerns

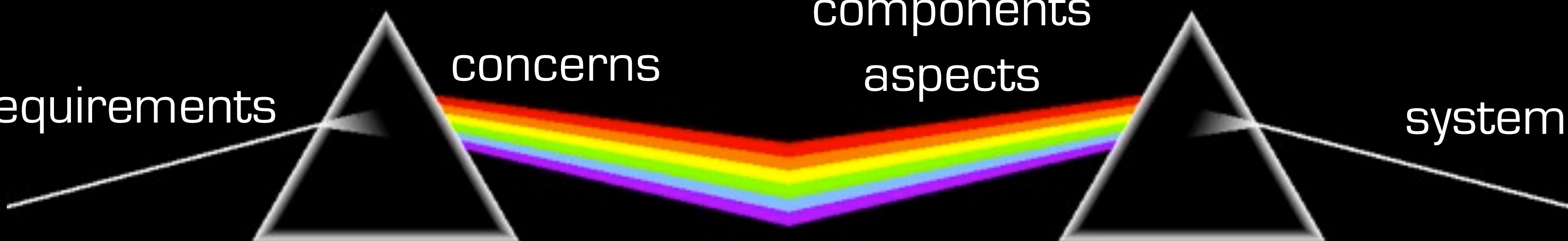
- Monitoring
- Security
- Coordination
- ...



What are Aspects?

Modular implementation of **crosscutting** concerns

Monitoring
Security
Coordination
...



one **goal**, different **mechanisms**

Pointcut / Advice

A novel programming language mechanism

- interesting in its own right!

Pointcut / Advice

A novel programming language mechanism

- interesting in its own right!

pointcut

Pointcut / Advice

A novel programming language mechanism

- interesting in its own right!

pointcut

advice

Pointcut / Advice

A novel programming language mechanism

- interesting in its own right!

join points

pointcut

advice

Pointcut / Advice

A novel programming language mechanism

- interesting in its own right!

join points

pointcut

advice

Pointcut / Advice

A novel programming language mechanism

- interesting in its own right!

join points

pointcut

advice

Pointcut / Advice

A novel programming language mechanism

- interesting in its own right!

join points

pointcut

advice

Pointcut / Advice

A novel programming language mechanism

- interesting in its own right!

join points

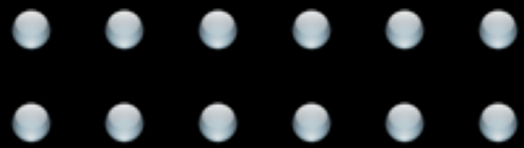
pointcut

advice

Pointcut / Advice

A novel programming language mechanism

- interesting in its own right!



join points

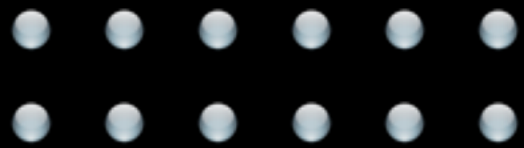
pointcut

advice

Pointcut / Advice

A novel programming language mechanism

- interesting in its own right!



join points

pointcut

advice

"glorification" of the observer pattern

Quantification

join points

pointcut

advice

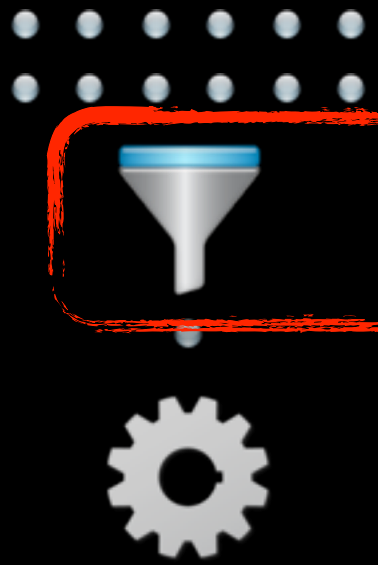
Quantification

join points

pointcut

advice

Quantification



join points

pointcut

```
execution(* Shape+.set*(..))
```

advice

Quantification


```
execution(* Shape+.set*(..))
&& this(s)
```

Quantification


```
pointcut change(Shape s):
   execution(* Shape+.set*(..))
   && this(s)
```

Quantification


```
pointcut change(Shape s):
   execution(* Shape+.set*(..))
   && this(s)
```

```
after(Shape s): change(s){
  // update observers
}
```

a join point

a join point

a join point

computation
inside!

a join point

computation
inside!

a join point

computation inside!

“around” advice can ignore it

a join point

computation
inside!

“around” advice can ignore it
or proceed

a join point

computation
inside!

“around” advice can ignore it
or proceed
and proceed…

a join point

computation
inside!

“around” advice can ignore it
or proceed
and proceed...

this is more than 1-way notifications

Why is this exciting?

Why is this exciting?

crosscutting is a real problem

Why is this exciting?

crosscutting is a real problem

pointcut/advice is effective for handling crosscutting

Why is this exciting?

crosscutting is a real problem

pointcut/advice is effective for handling crosscutting

- behavioral reflection for mere mortals

Why is this exciting?

crosscutting is a real problem

pointcut/advice is effective for handling crosscutting

- behavioral reflection for mere mortals
- more declarative, esp. wrt **quantification** (pointcuts)

Why is this exciting?

crosscutting is a real problem

pointcut/advice is effective for handling crosscutting

- behavioral reflection for mere mortals
- more declarative, esp. wrt **quantification** (pointcuts)
- more amenable to analysis (or so it seems)

Why is this exciting?

crosscutting is a real problem

pointcut/advice is effective for handling crosscutting

- behavioral reflection for mere mortals
- more declarative, esp. wrt **quantification** (pointcuts)
- more amenable to analysis (or so it seems)

still not there yet

- lots of open challenges

State of the Practice

State of the Practice

- every execution step is a join point

State of the Practice

- every execution step is a join point
- pointcuts “see” them all

State of the Practice

- every execution step is a join point
- pointcuts “see” them all
- advice can do anything

State of the Practice

- every execution step is a join point
- pointcuts "see" them all
- advice can do anything
  - proceed 0..n times

State of the Practice

- every execution step is a join point
- pointcuts “see” them all
- advice can do anything
  - proceed 0..n times
  - change arguments, return value

State of the Practice

- every execution step is a join point
- pointcuts “see” them all
- advice can do anything
  - proceed 0..n times
  - change arguments, return value
  - arbitrary side effects

FEATURE	APPLICATION

FEATURE	APPLICATION
all execution steps are join points, pointcuts see them all	

FEATURE	APPLICATION
all execution steps are join points, pointcuts see them all	unanticipated evolution, “obliviousness”

FEATURE	APPLICATION
all execution steps are join points, pointcuts see them all	unanticipated evolution, “obliviousness”
advice that does not proceed	

FEATURE	APPLICATION
all execution steps are join points, pointcuts see them all	unanticipated evolution, “obliviousness”
advice that does not proceed	memoization, proxies, ...

FEATURE	APPLICATION
all execution steps are join points, pointcuts see them all	unanticipated evolution, "obliviousness"
advice that does not proceed	memoization, proxies, ...
advice that proceeds n times	

FEATURE	APPLICATION
all execution steps are join points, pointcuts see them all	unanticipated evolution, “obliviousness”
advice that does not proceed	memoization, proxies, ...
advice that proceeds n times	retry, redundancy, ...

FEATURE	APPLICATION
all execution steps are join points, pointcuts see them all	unanticipated evolution, “obliviousness”
advice that does not proceed	memoization, proxies, ...
advice that proceeds n times	retry, redundancy, ...
changing arguments/return	

FEATURE	APPLICATION
all execution steps are join points, pointcuts see them all	unanticipated evolution, “obliviousness”
advice that does not proceed	memoization, proxies, ...
advice that proceeds n times	retry, redundancy, ...
changing arguments/return	encryption, comfort zone, ...

FEATURE	APPLICATION
all execution steps are join points, pointcuts see them all	unanticipated evolution, "obliviousness"
advice that does not proceed	memoization, proxies, ...
advice that proceeds n times	retry, redundancy, ...
changing arguments/return	encryption, comfort zone, ...
arbitrary side effects	

FEATURE	APPLICATION
all execution steps are join points, pointcuts see them all	unanticipated evolution, “obliviousness”
advice that does not proceed	memoization, proxies, ...
advice that proceeds n times	retry, redundancy, ...
changing arguments/return	encryption, comfort zone, ...
arbitrary side effects	almost all aspects!

BUT...

BUT…

```
void around(): call(int Fib.calc(int)){
   System.out = myPrivateStream;
   return -1;
}
```

Break semantics!

BUT...

```
void around(): call(int Fib.calc(int)){
   System.out = myPrivateStream;
   return -1;
}
```

Break semantics!

```
void around(): call(void SecurityManager.check*(..)){}
```

No more security!

BUT...

```
void around(): call(int Fib.calc(int)){
  System.out = myPrivateStream;
  return -1;
}
```

Break semantics!

```
void around(): call(void SecurityManager.check*(..)){}
```

No more security!

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

ClassCastException!

BUT...

```
void around(): call(int Fib.calc(int)){
   System.out = myPrivateStream;
   return -1;
}
```

Break semantics!

```
void around(): call(void SecurityManager.check*(..)){}
```

No more security!

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

ClassCastException!

```
before(Person p): execution(* *(..)) && this(p) {
   System.out.println("person active: " + p.getName());
}
```

StackOverflow!

BUT…

```
void around(): call(int Fib.calc(int)){
   System.out = myPrivateStream;
   return -1;
}
```

Break semantics!

```
void around(): call(void SecurityManager.check*(..)){}
```

No more security!

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

ClassCastException!

```
before(Person p): execution(* *(..)) && this(p) {
  System.out.println("person active: " + p.getName());
}
```

StackOverflow!

Aspect

Orientation

ASPECT

ORIENTATION

ASPECT

??

ORIENTATION

ASPECT ORIENTATION

ASPECT

ORIENTATION

TAMING

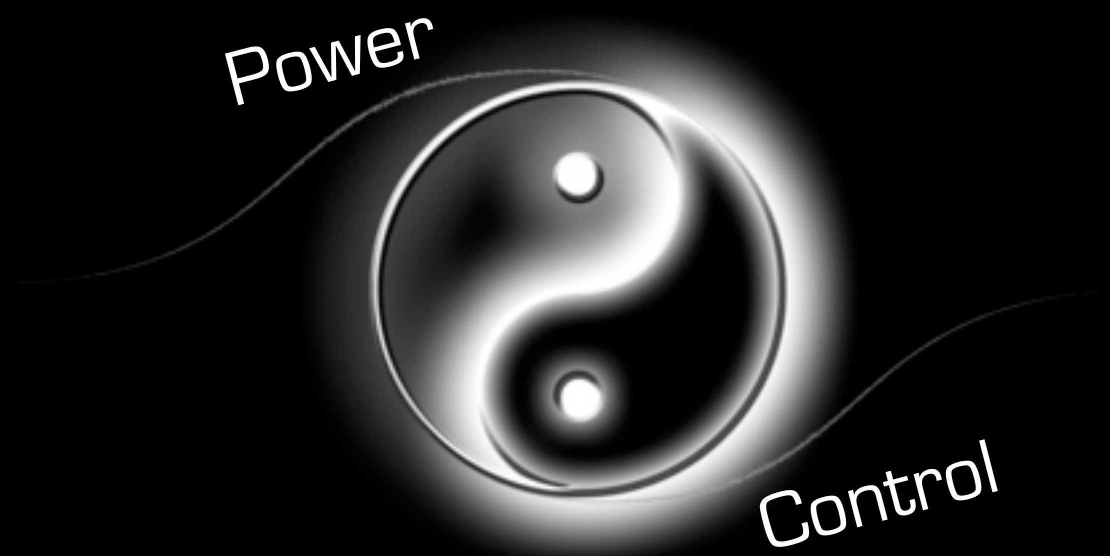
ASPECT

ORIENTATION

TAMING

ASPECT

ORIENTATION



Scoping

Interfaces

Types

Effects

Scoping

Interfaces

Types

Effects

Dynamic

Static

Scoping

Interfaces

Types

Effects

Scoping

Can we restrict quantification to well-defined boundaries?

What abstractions are meaningful?

Interfaces

Types

Effects

Global Quantification

Global Quantification

Global visibility of join points exacerbates many issues

Global Quantification

Global visibility of join points exacerbates many issues

- accidental matches

Global Quantification

Global visibility of join points exacerbates many issues

- accidental matches
- spurious interferences

Global Quantification

Global visibility of join points exacerbates many issues

- accidental matches
- spurious interferences
- advice loops

Global Quantification

Global visibility of join points exacerbates many issues

- accidental matches
- spurious interferences
- advice loops
- etc.

Mitigating the Issue

Mitigating the Issue

Explicit announcement of join points

- explicit join points [Hoffman, 2012]
- quantified typed events [Rajan, 2008]
- closure join points [Bodden, 2011]
- open applications
- etc.

Mitigating the Issue

Explicit announcement of join points

- explicit join points [Hoffman, 2012]
- quantified typed events [Rajan, 2008]
- closure join points [Bodden, 2011]
- open applications
- etc.

Expressive pointcuts

- rich pointcuts for robust patterns [Gybels, 2003], [Ostermann, 2005]
- application-specific pointcuts [Brichau, 2008]
- annotations [Kiczales, 2005]
- etc.

Scoped Quantification

Scoped Quantification

Global quantification

- just as bad as global mutable variables!

Scoped Quantification

Global quantification

- just as bad as global mutable variables!

Different scoping disciplines for identifiers

- lexical scope
- dynamic scope
- thread-local
- per object, class, module

Scoped Quantification

Global quantification

- just as bad as global mutable variables!

Different scoping disciplines for identifiers

- lexical scope
- dynamic scope
- thread-local
- per object, class, module

All have been explored for aspects as well

- CaesarJ, AspectScheme, Eos, AspectJ...

Scoped Quantification: Advanced Models

Scoped Quantification: Advanced Models

Scoping strategies [Tanter, 2008/2009/2010a]

- killer app: access control [Toledo, 2011/ 2012/2013]

Execution levels [Tanter, 2010b]

Membranes [Tanter, 2012] [Figueroa, 2013]

Scoped Quantification: Advanced Models

Scoping strategies [Tanter, 2008/2009/2010a]

- killer app: access control [Toledo, 2011/ 2012/2013]

Execution levels [Tanter, 2010b]

Membranes [Tanter, 2012] [Figueroa, 2013]

Scoped Quantification: Advanced Models

Scoping strategies [Tanter, 2008/2009/2010a]

- killer app: access control [Toledo, 2011/ 2012/2013]

Execution levels [Tanter, 2010b]

Membranes [Tanter, 2012] [Figueroa, 2013]

Execution Levels

Composing Dynamic Analyses

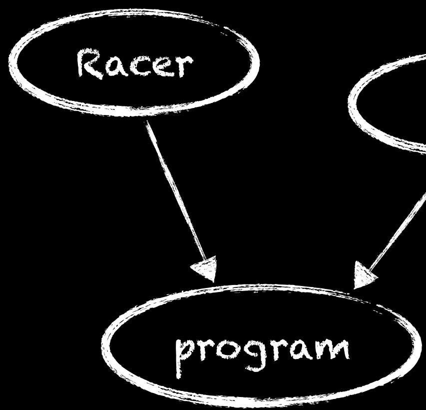
joint work with
Walter Binder & co

[Tanter, 2010c]

Composing Dynamic Analyses

joint work with
Walter Binder & co

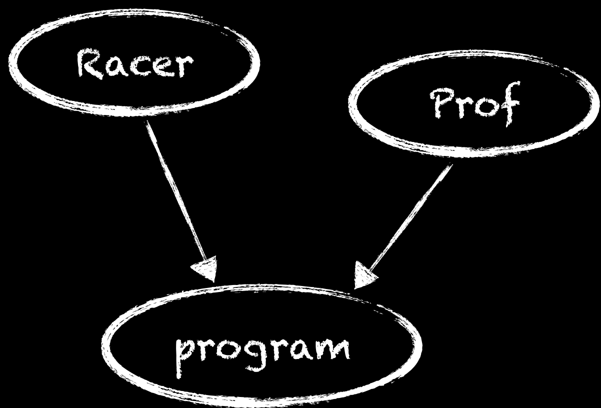
[Tanter, 2010c]

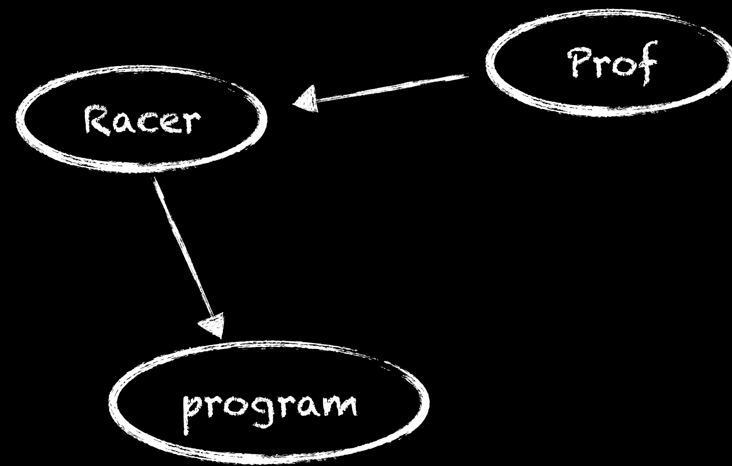


Composing Dynamic Analyses

joint work with
Walter Binder & co

[Tanter, 2010c]

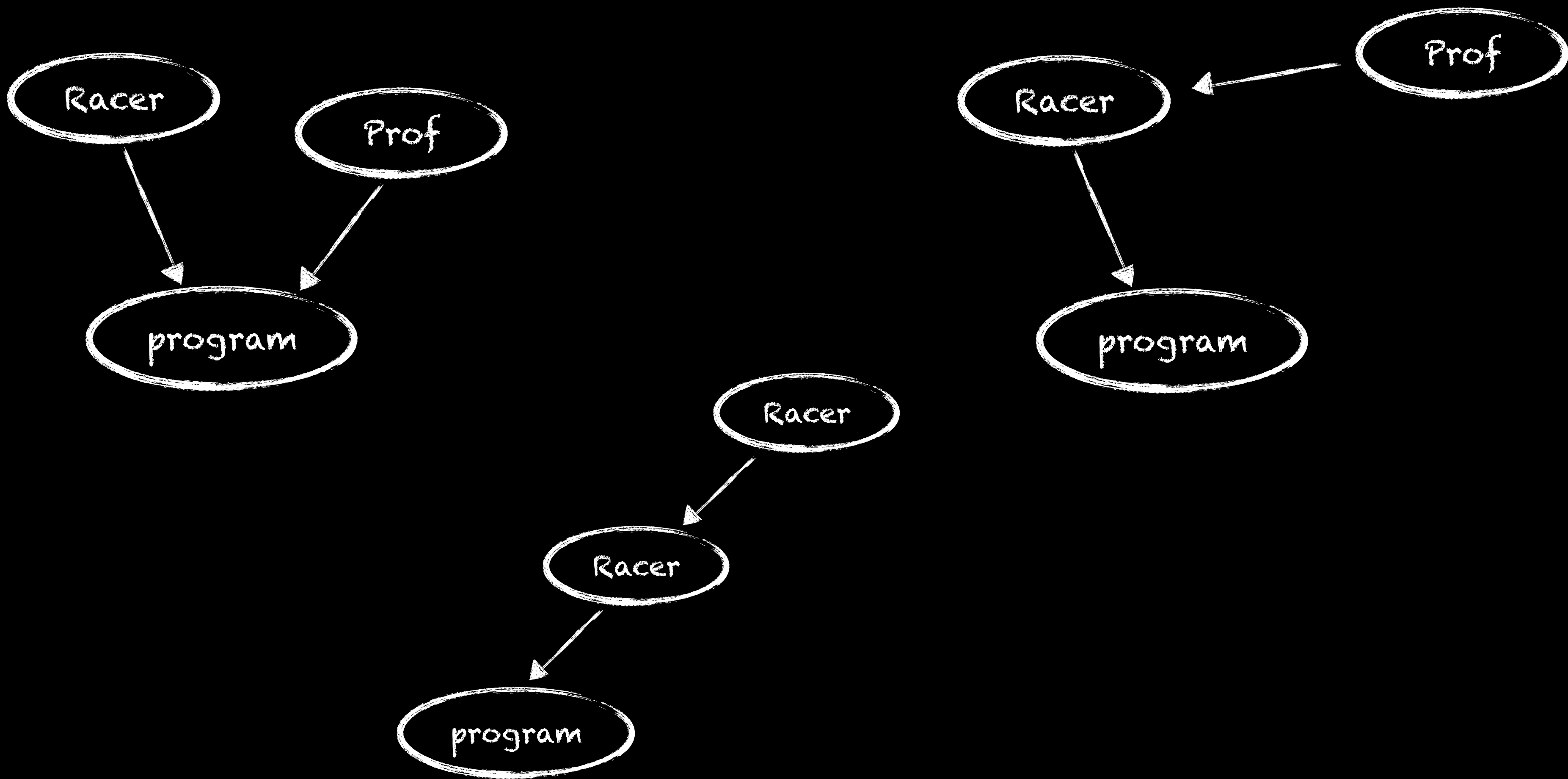




Composing Dynamic Analyses

joint work with
Walter Binder & co

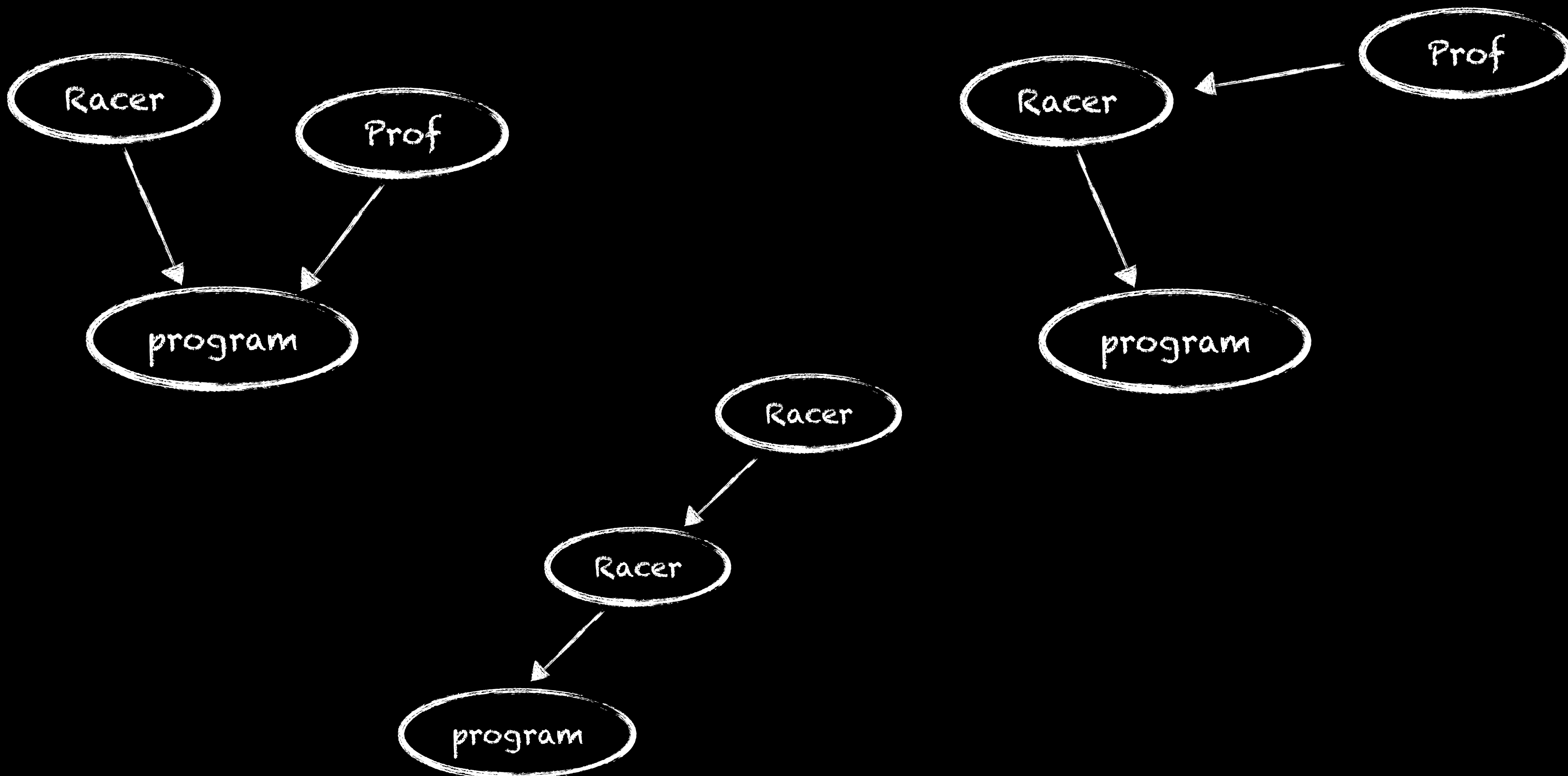
[Tanter, 2010c]



Composing Dynamic Analyses

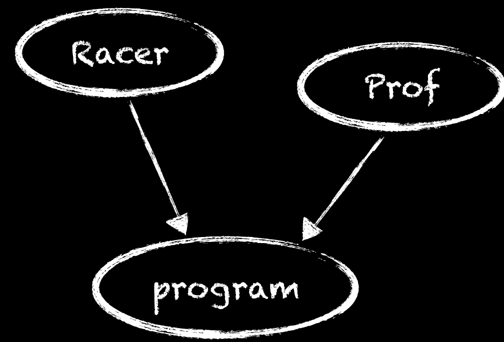
joint work with
Walter Binder & co

[Tanter, 2010c]



NONE CAN BE IMPLEMENTED! (until now...)

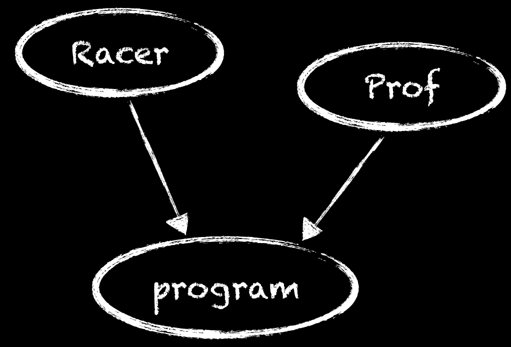
Why?



Why?

Each aspect alters the observation of others

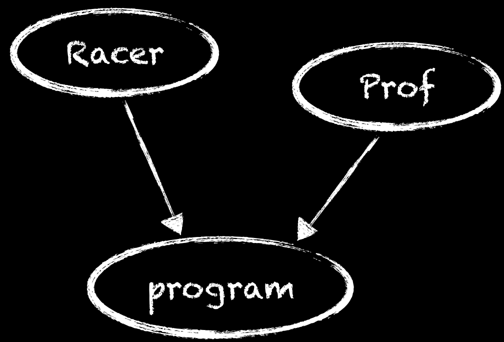
- Racer creates objects
- Prof accesses fields



WHY?

Each aspect alters the observation of others

- Racer creates objects
- Prof accesses fields



Each aspect potentially sees itself

- infinite regression

Execution levels

[Tanter, 2010b]

Structure computation in levels

- aspects stand at specific levels
- observe computation below

Execution levels

[Tanter, 2010b]

Structure computation in levels

- aspects stand at specific levels
- observe computation below

level 0

Execution levels

[Tanter, 2010b]

Structure computation in levels

- aspects stand at specific levels
- observe computation below

level 1

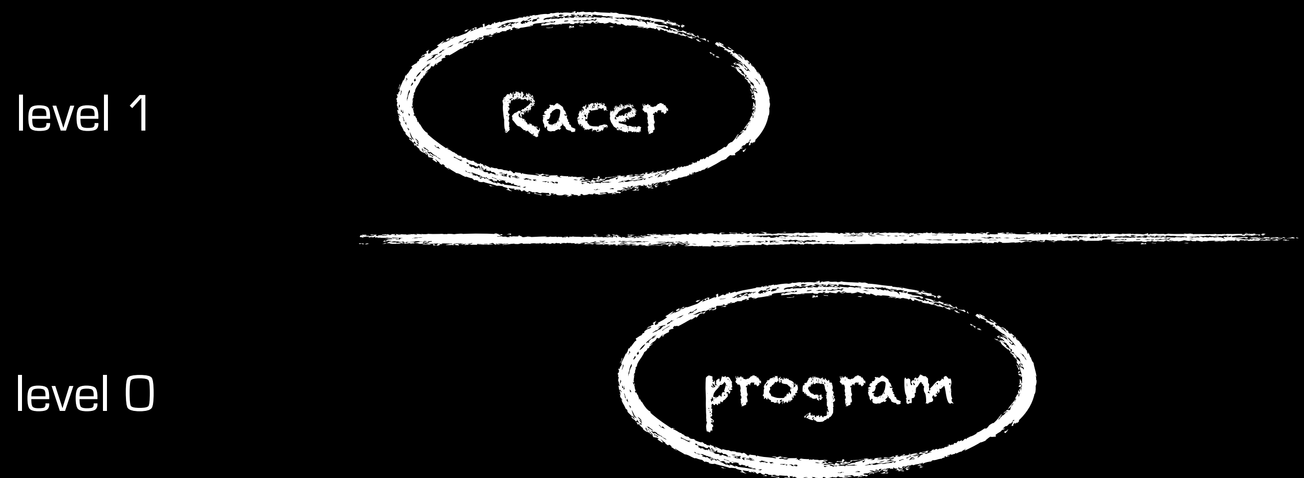
level 0

Execution levels

[Tanter, 2010b]

Structure computation in levels

- aspects stand at specific levels
- observe computation below

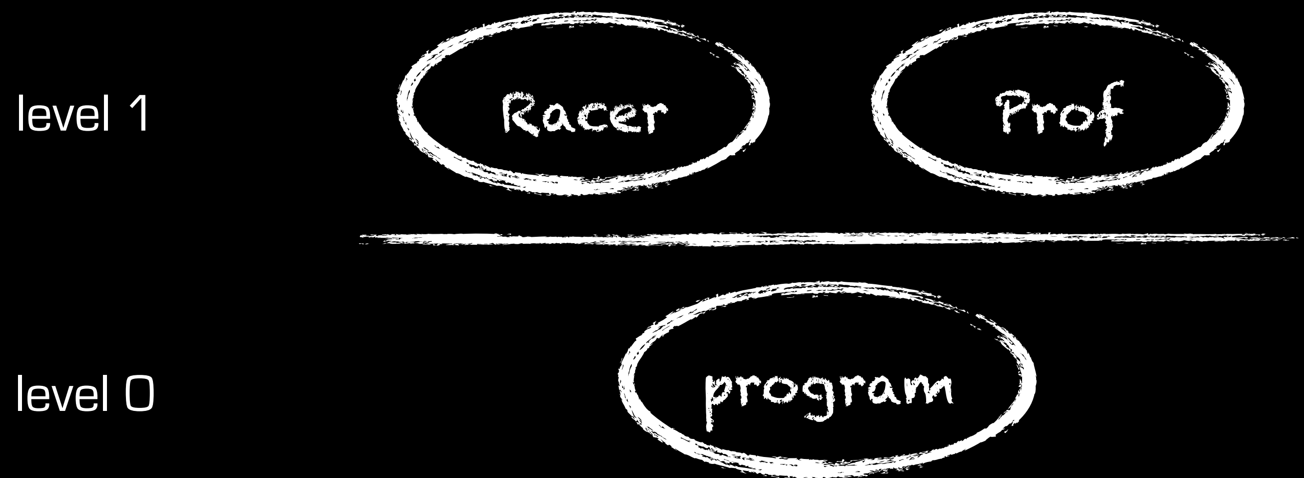


Execution levels

[Tanter, 2010b]

Structure computation in levels

- aspects stand at specific levels
- observe computation below

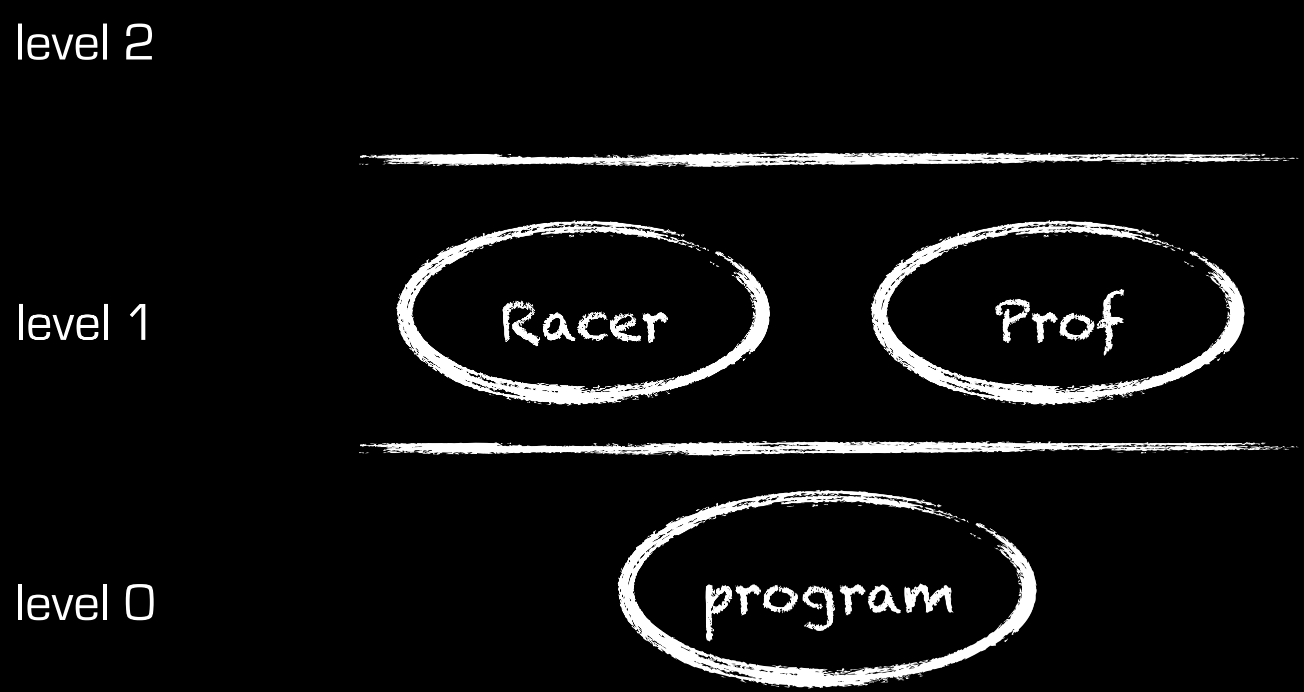


Execution levels

[Tanter, 2010b]

Structure computation in levels

- aspects stand at specific levels
- observe computation below

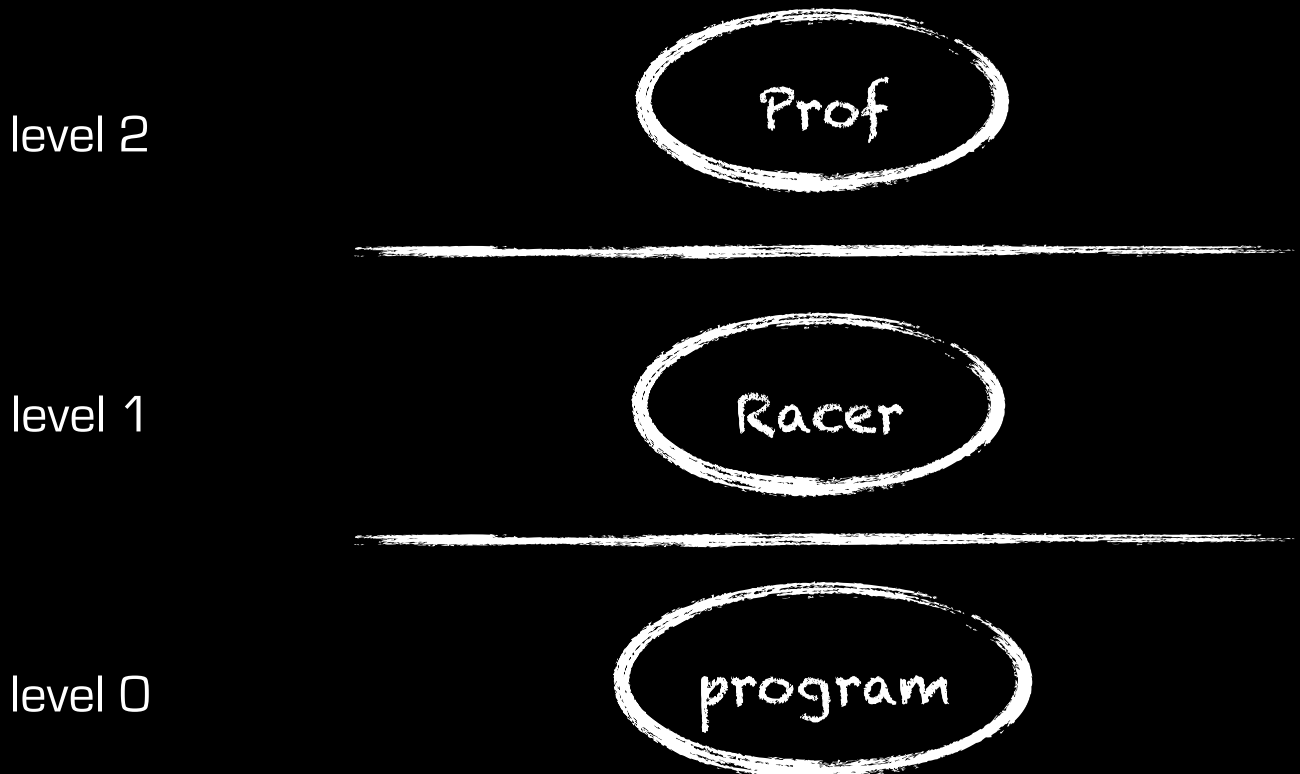


Execution levels

[Tanter, 2010b]

Structure computation in levels

- aspects stand at specific levels
- observe computation below

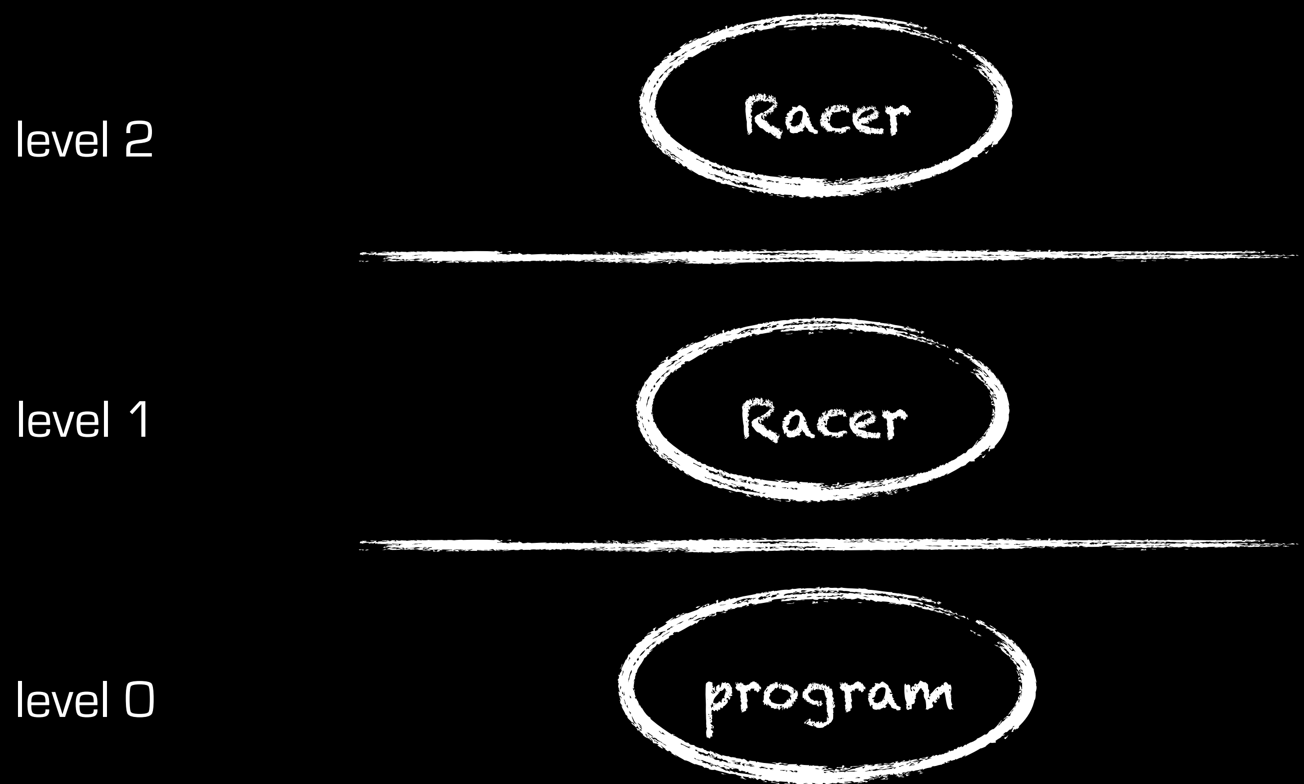


Execution levels

[Tanter, 2010b]

Structure computation in levels

- aspects stand at specific levels
- observe computation below



Levels: Theory and Practice

Levels: Theory and Practice

Strong guarantee: aspect loops are avoided

joint work with

Nicolas Tabareau

Ismael Figueroa

Levels: Theory and Practice

Strong guarantee: aspect loops are avoided

Can be implemented efficiently [Tanter, 2010c; Moret, 2011]

joint work with

Walter Binder

Philippe Moret, Danilo Ansaloni

Levels: Theory and Practice

Strong guarantee: aspect loops are avoided

Can be implemented efficiently [Tanter, 2010c; Moret, 2011]

Ad-hoc checks in practice

- 1/3 of all aspects in the "AspectJ in Action" book
- 18% of all pointcuts in corpus of ≈500 aspects
- all aspects work out-of-the-box with default level semantics

Topological Scoping

Topological Scoping

Execution levels

- give structure to computation
- use this structure to define scoping
- come with some properties (eg. no loop)

Topological Scoping

Execution levels

- give structure to computation
- use this structure to define scoping
- come with some properties (eg. no loop)

This is an example of topological scoping

- topology: tower
- what about others?

Topological Scoping

Execution levels

- give structure to computation
- use this structure to define scoping
- come with some properties (eg. no loop)

This is an example of topological scoping

- topology: tower
- what about others?

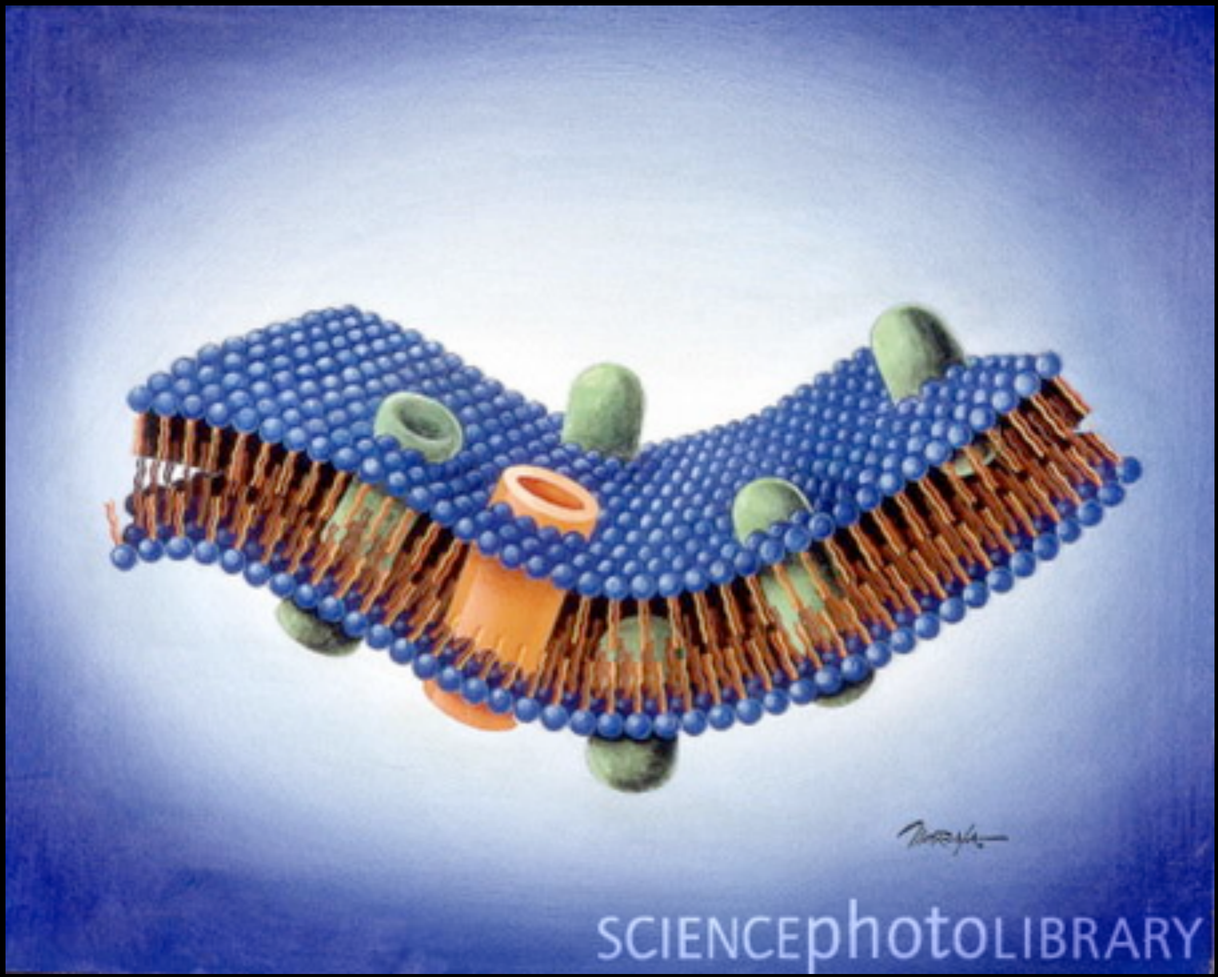
Membranes for AOP

joint work with
Nicolas Tabareau
Rémi Douence
Ismael Figueroa

GIVING STRUCTURE TO COMPUTATION

Programmable membranes [Boudol, 2004; Schmitt, 2004]

- inspired by membranes in biology



Membranes for AOP

[Tanter, 2012]

Why not use membranes for AOP?

- gives rise to flexible topological scoping
- supports control over certain effects

Membranes for AOP

[Tanter, 2012]

Why not use membranes for AOP?

- gives rise to flexible topological scoping
- supports control over certain effects

X

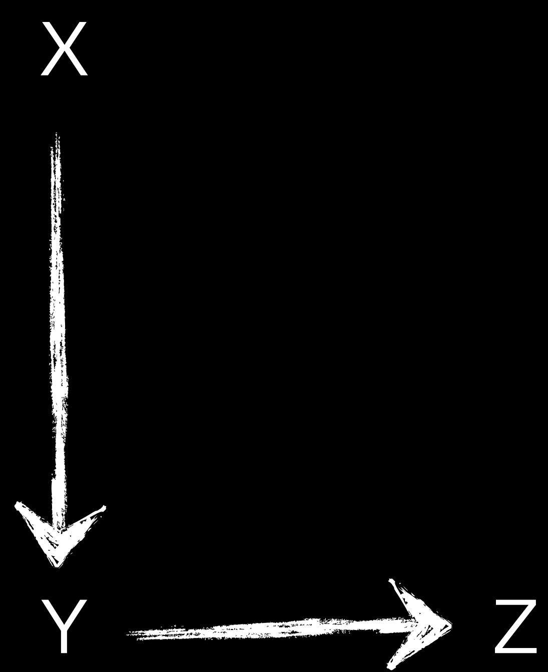
Y

Membranes for AOP

[Tanter, 2012]

Why not use membranes for AOP?

- gives rise to flexible topological scoping
- supports control over certain effects

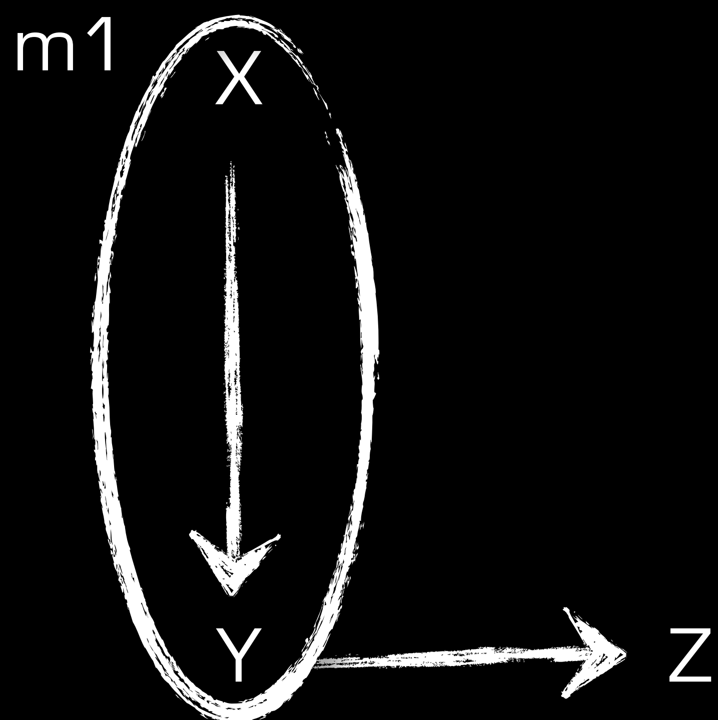


Membranes for AOP

[Tanter, 2012]

Why not use membranes for AOP?

- gives rise to flexible topological scoping
- supports control over certain effects

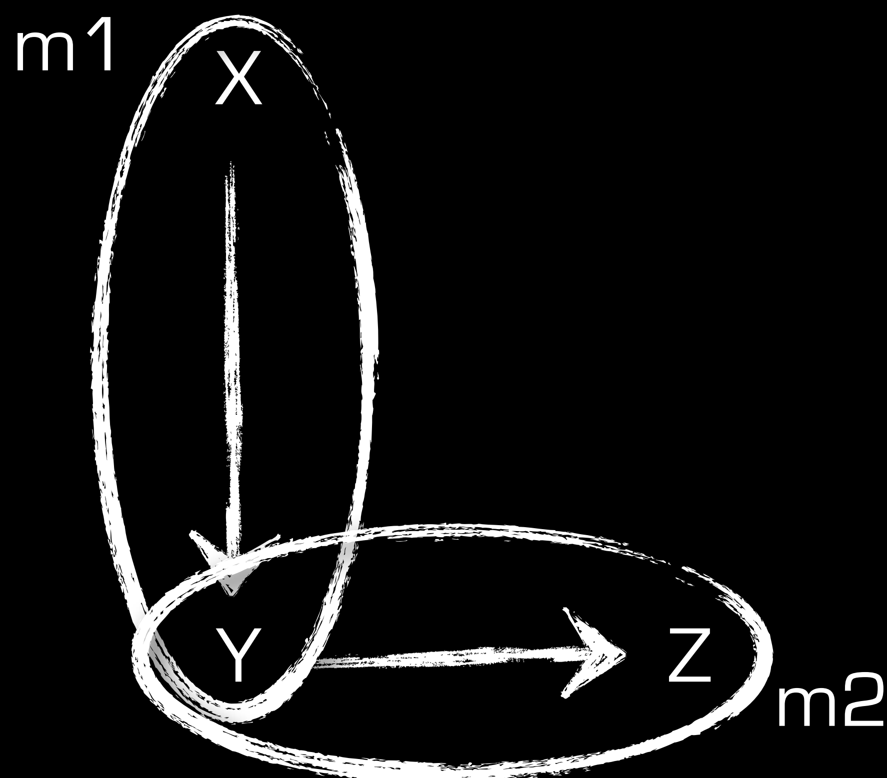


Membranes for AOP

[Tanter, 2012]

Why not use membranes for AOP?

- gives rise to flexible topological scoping
- supports control over certain effects

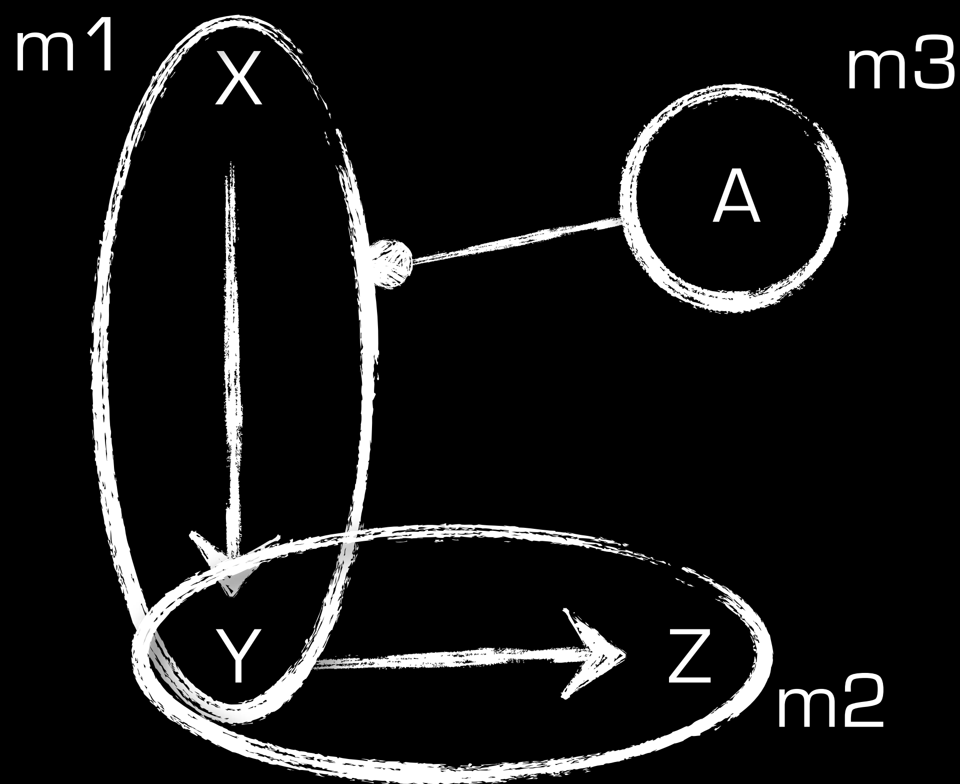


Membranes for AOP

[Tanter, 2012]

Why not use membranes for AOP?

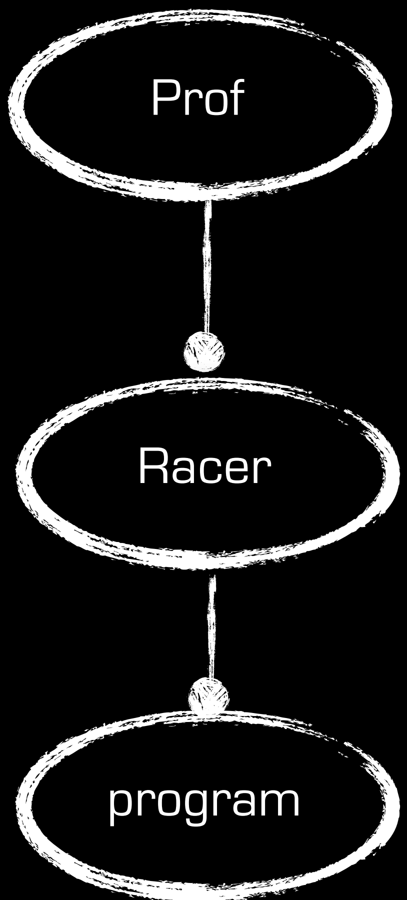
- gives rise to flexible topological scoping
- supports control over certain effects



Wednesday, April 17, 13

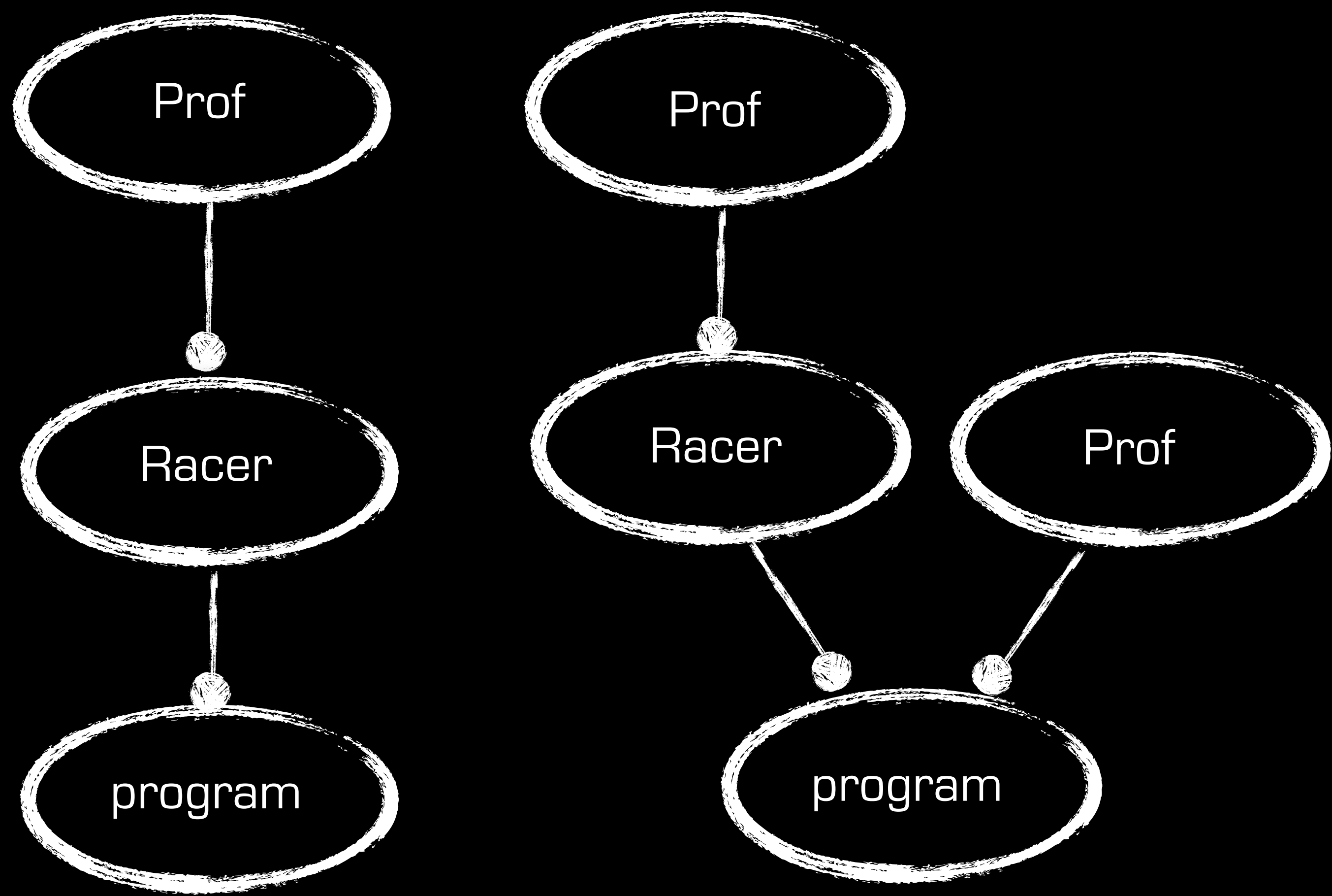
Topological Scoping with Membranes

Topological Scoping with Membranes



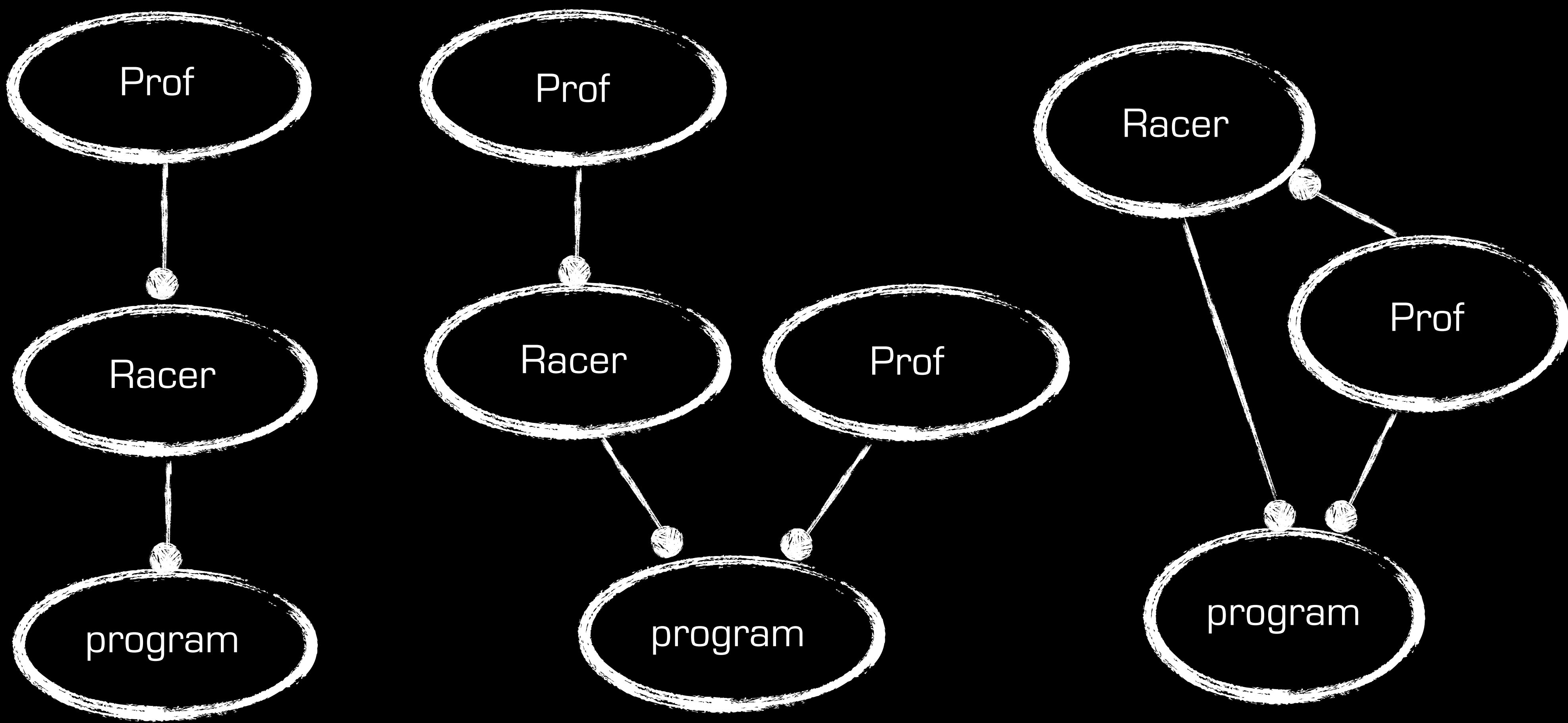
execution levels

Topological Scoping with Membranes



execution levels tree

Topological Scoping with Membranes



execution levels tree DAG

Membranes: Theory and Practice

[Figueroa, 2013]

Membranes: Theory and Practice

Wide design space

- how to create, deploy and configure membranes?
- can membranes crosscut? organized hierarchically?
- what guarantees are expected? tradeoff?
- MAScheme, AO Haskell [Figueroa, 2013]

Membranes: Theory and Practice

Wide design space

- how to create, deploy and configure membranes?
- can membranes crosscut? organized hierarchically?
- what guarantees are expected? tradeoff?
- MAScheme, AO Haskell [Figueroa, 2013]

Exploit programmability

- ensure safety properties
- what language is useful to program membranes?
- Kell calculus

Scoping

Interfaces

Types

Effects

Scoping

Interfaces

Types

Effects

Can we reconcile quantification with modular reasoning?

What kind of static interfaces allow independent development?

Issues with Pointcut/Advice

class A

class B

class C

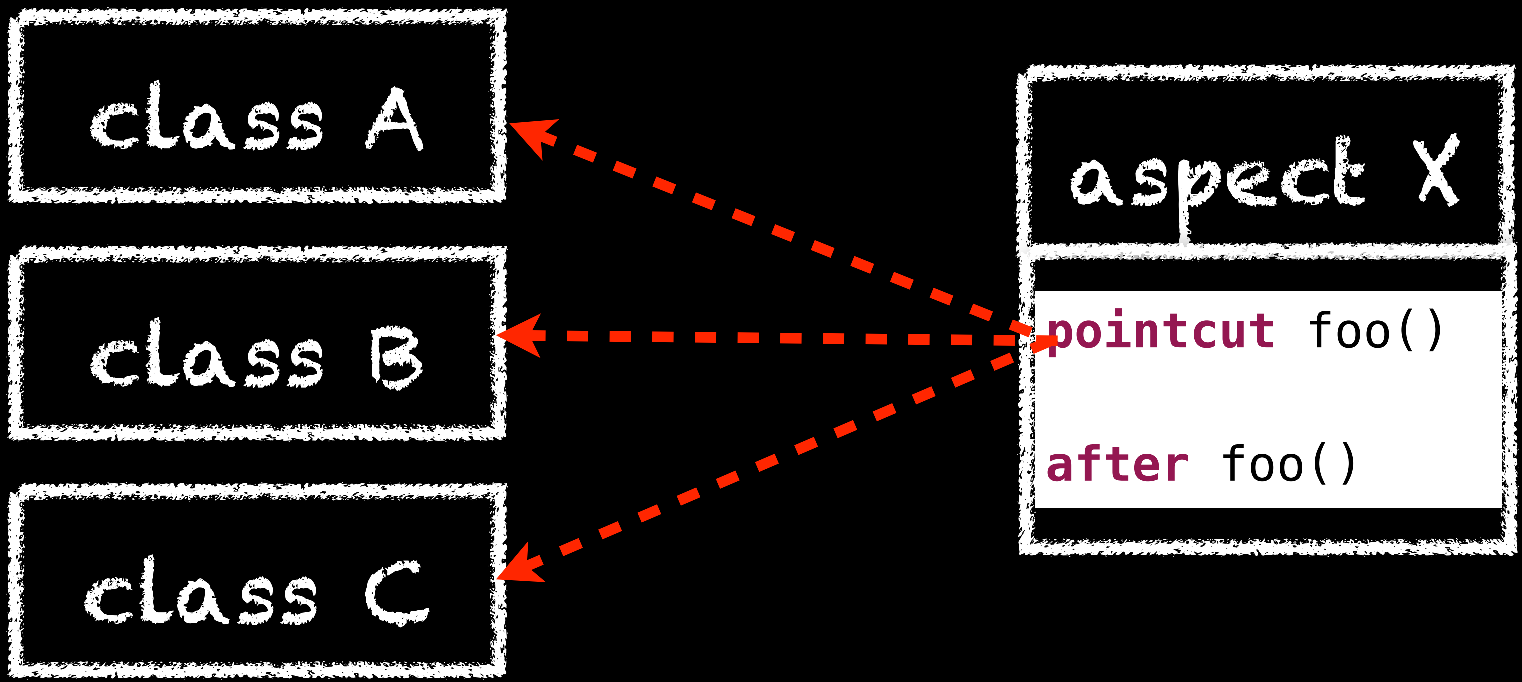
aspect X

```
pointcut foo()

after foo()
```

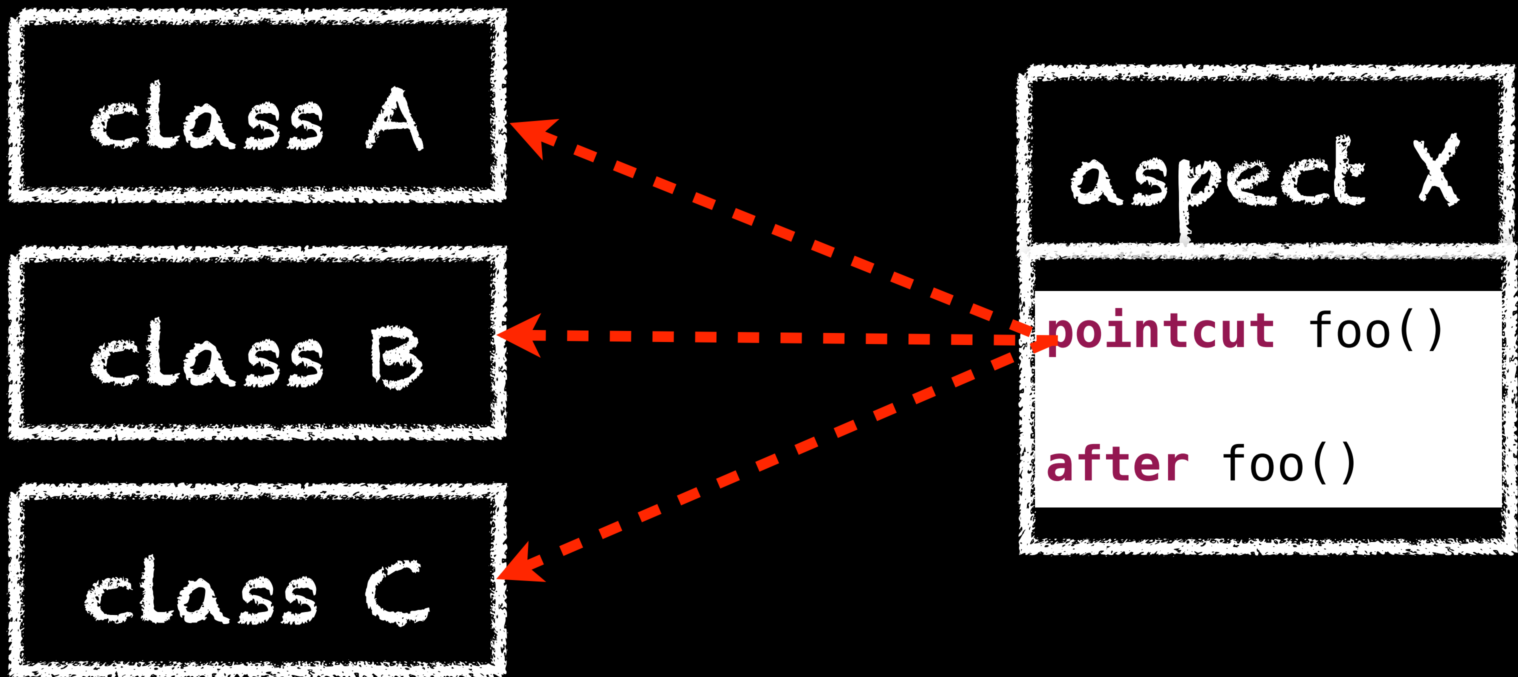
Issues with Pointcut/Advice

fragile dependencies



Issues with Pointcut/Advice

fragile dependencies

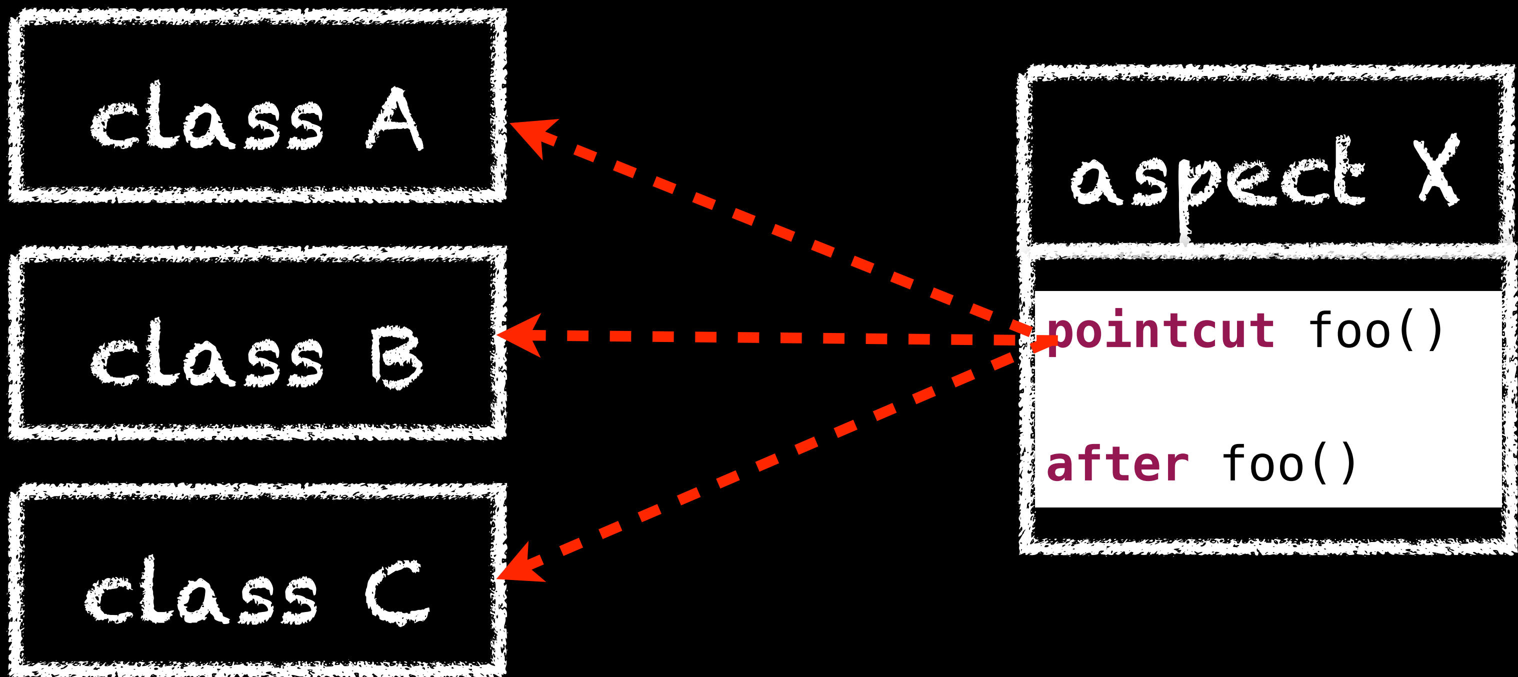


oblivious

⇒ no idea what is relied upon

Issues with Pointcut/Advice

fragile dependencies



oblivious

⇒ no idea what is relied upon

modular reasoning?

independent development?

Modular Reasoning?

Modular Reasoning?

Kiczales & Mezini [2005]

- fundamental issue is the crosscutting nature
- AOP makes the crosscutting concern explicit

Modular Reasoning?

Kiczales & Mezini [2005]

- fundamental issue is the crosscutting nature
- AOP makes the crosscutting concern explicit

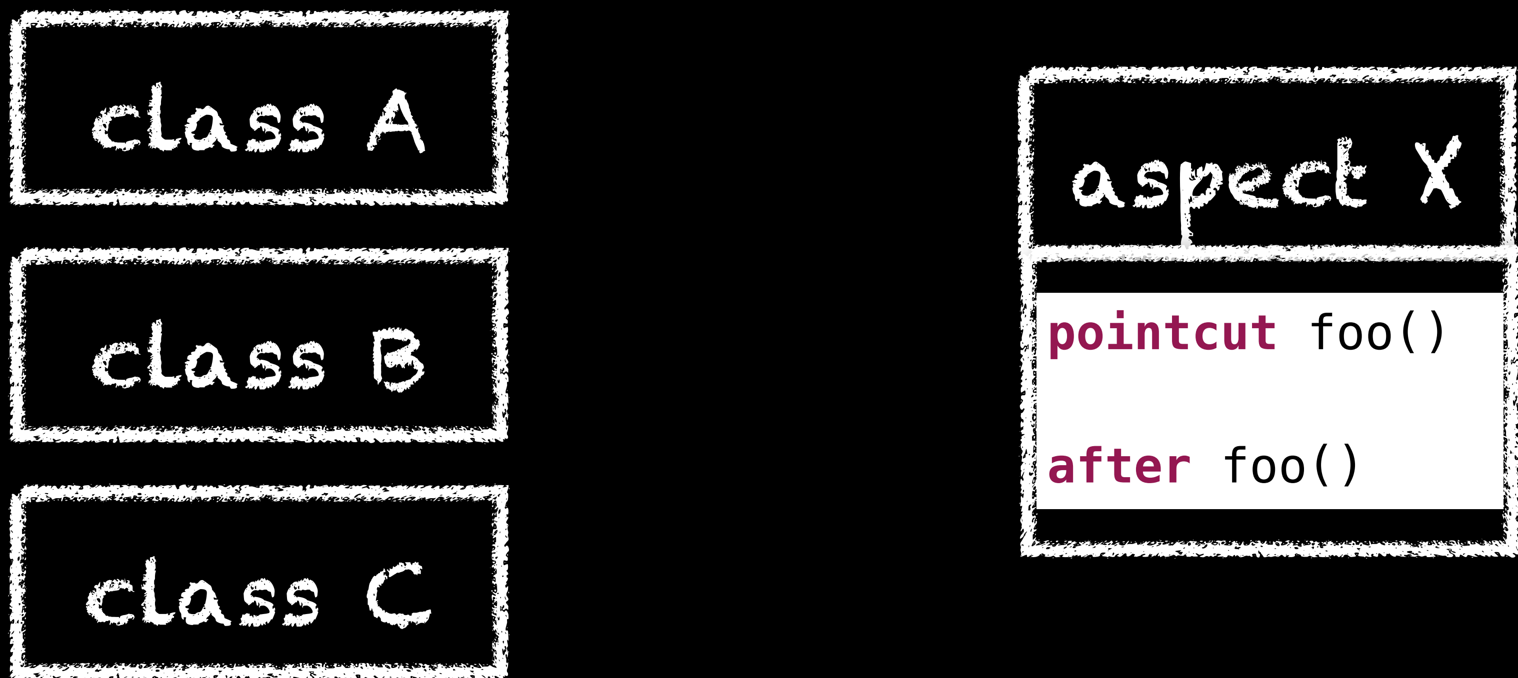
aspect-aware interfaces

Modular Reasoning?

Kiczales & Mezini [2005]

- fundamental issue is the crosscutting nature
- AOP makes the crosscutting concern explicit

aspect-aware interfaces

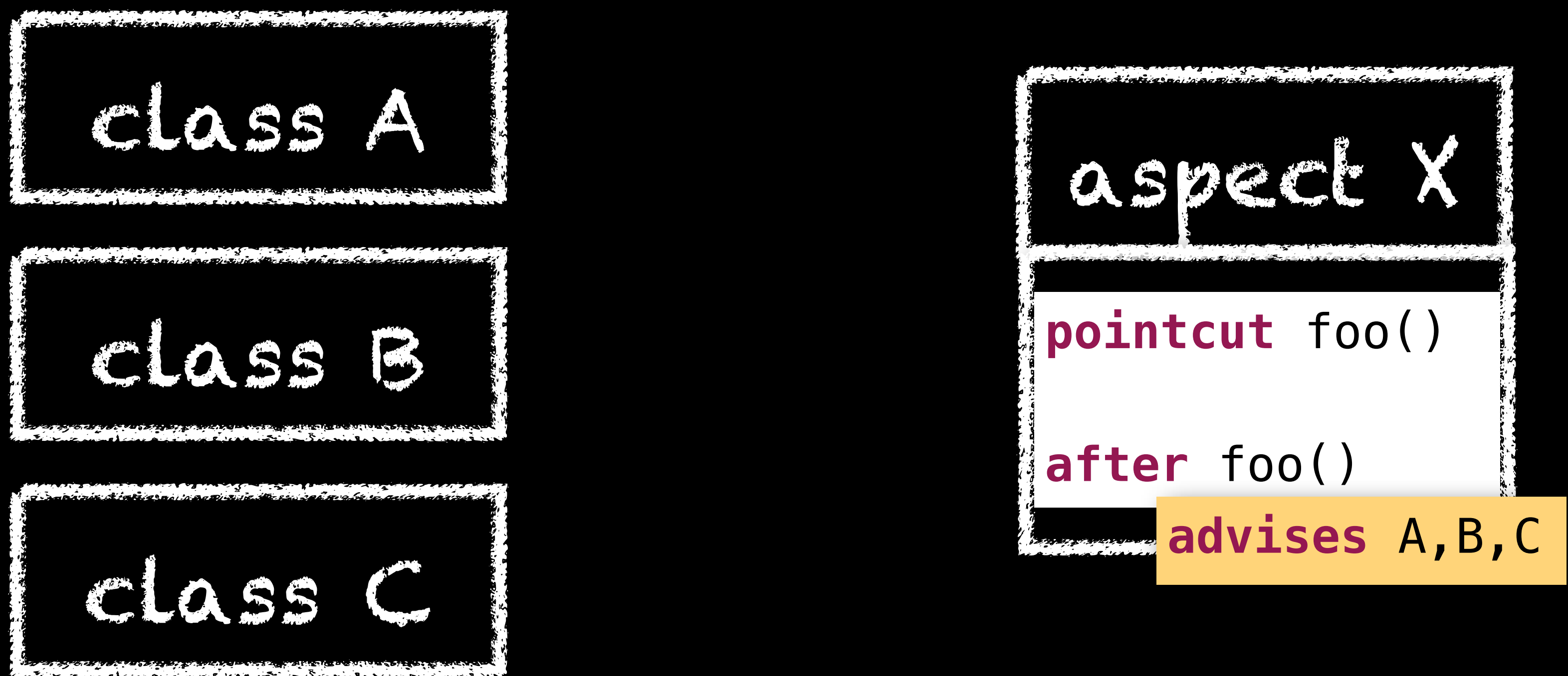


Modular Reasoning?

Kiczales & Mezini [2005]

- fundamental issue is the crosscutting nature
- AOP makes the crosscutting concern explicit

aspect-aware interfaces

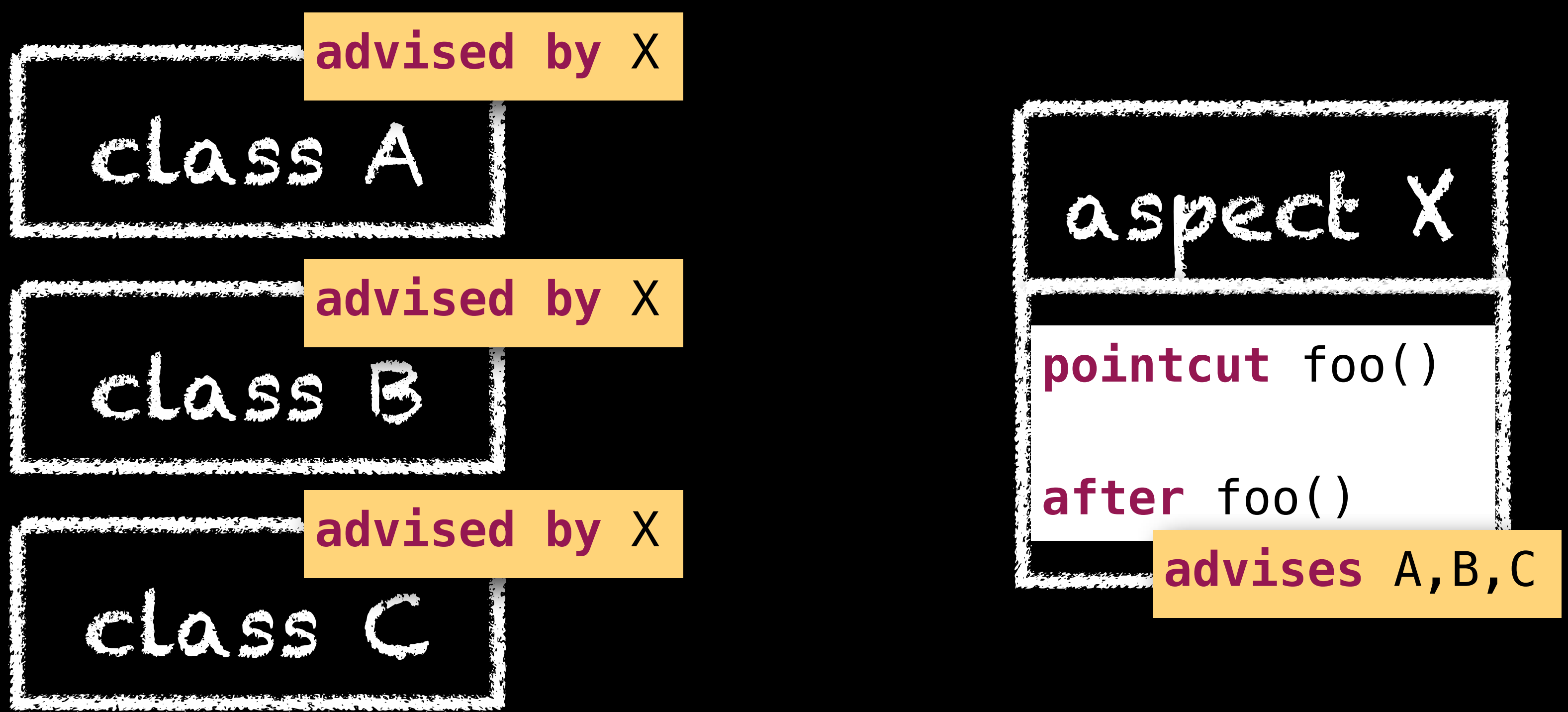


Modular Reasoning?

Kiczales & Mezini [2005]

- fundamental issue is the crosscutting nature
- AOP makes the crosscutting concern explicit

aspect-aware interfaces

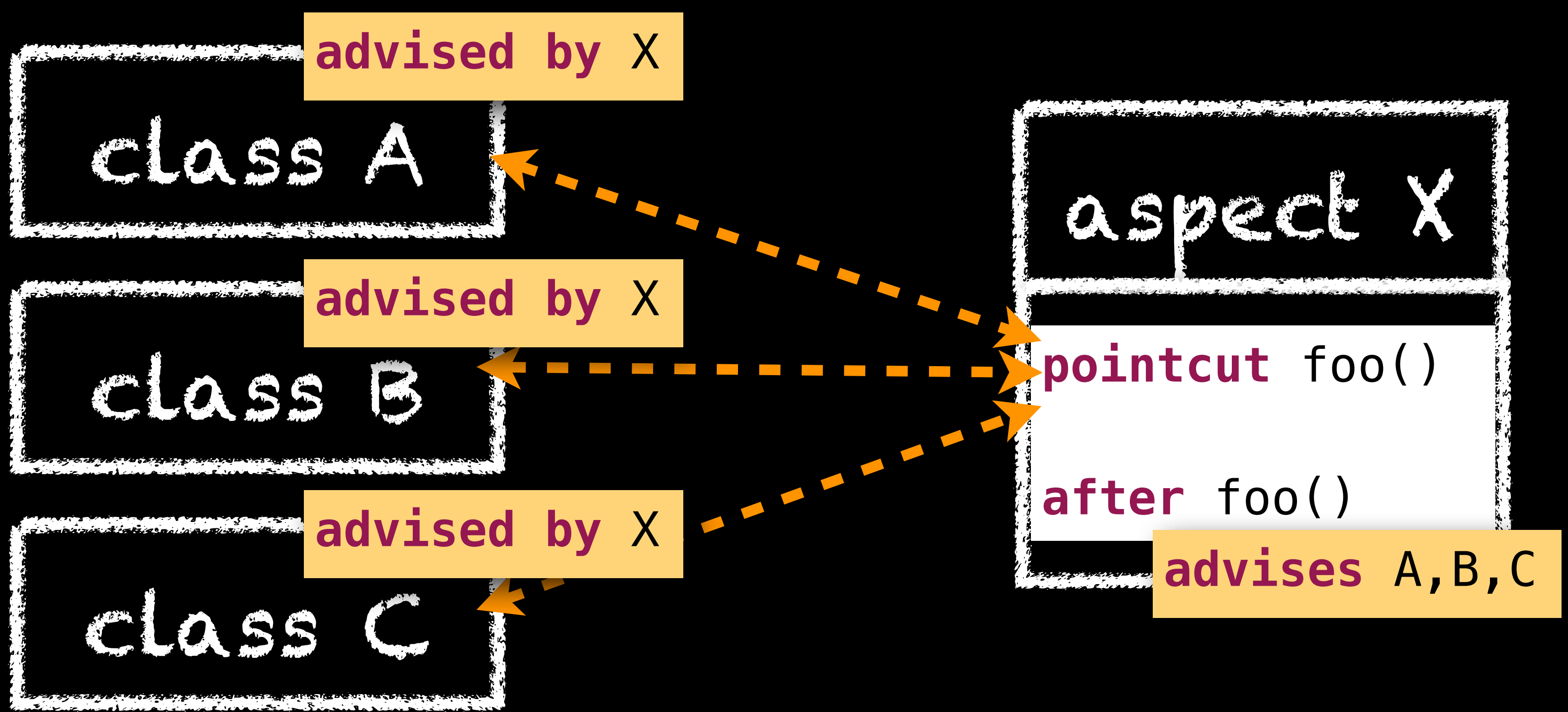


Modular Reasoning?

Kiczales & Mezini [2005]

- fundamental issue is the crosscutting nature
- AOP makes the crosscutting concern explicit

aspect-aware interfaces



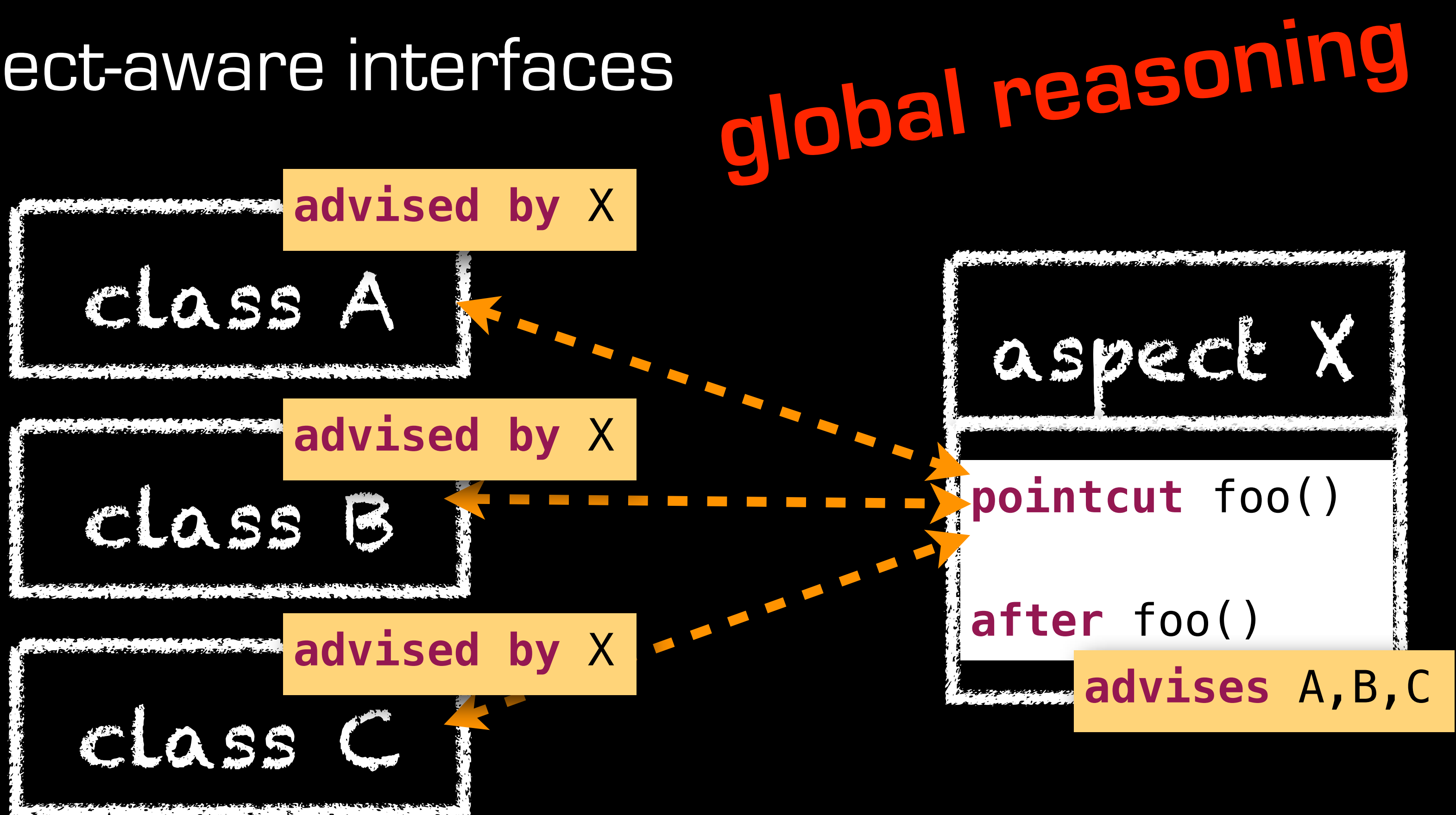
explicit dependencies

Modular Reasoning?

Kiczales & Mezini [2005]

- fundamental issue is the crosscutting nature
- AOP makes the crosscutting concern explicit

aspect-aware interfaces



explicit dependencies

Recovering Modular Reasoning

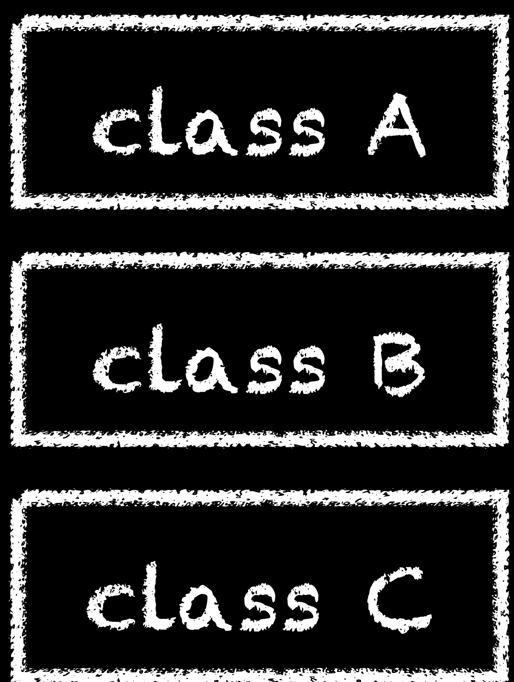
Putting pointcuts in interfaces [Gudmundson, 2001]

- Open Modules [Aldrich, 2005], etc.

Recovering Modular Reasoning

Putting pointcuts in interfaces [Gudmundson, 2001]

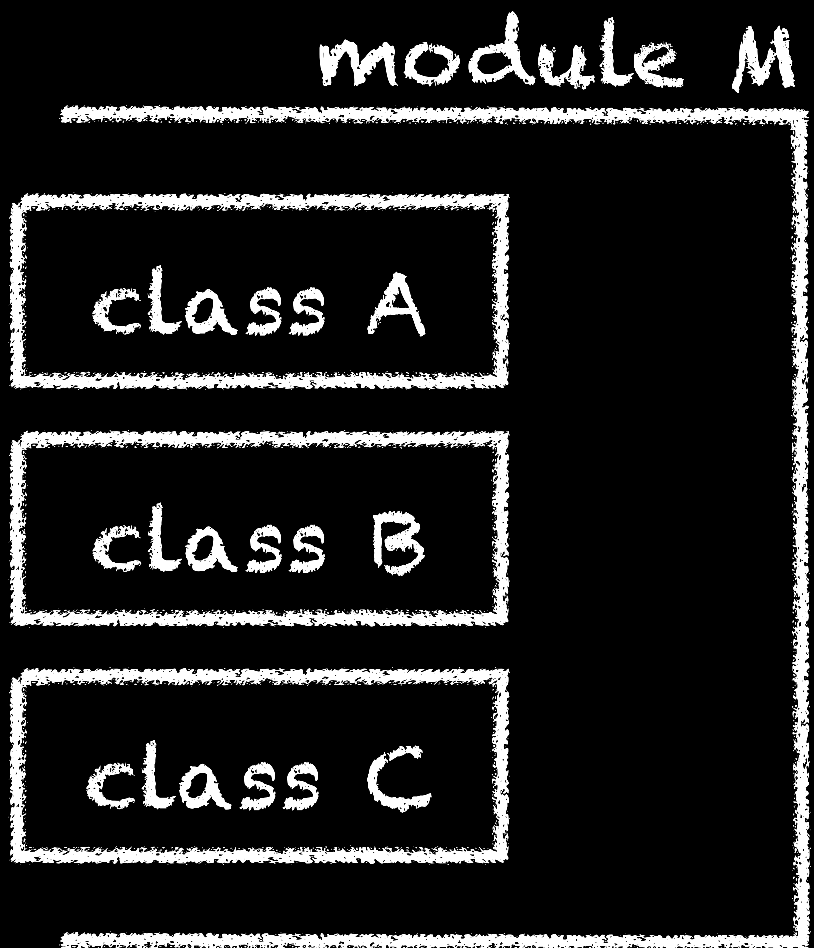
- Open Modules [Aldrich, 2005], etc.



Recovering Modular Reasoning

Putting pointcuts in interfaces [Gudmundson, 2001]

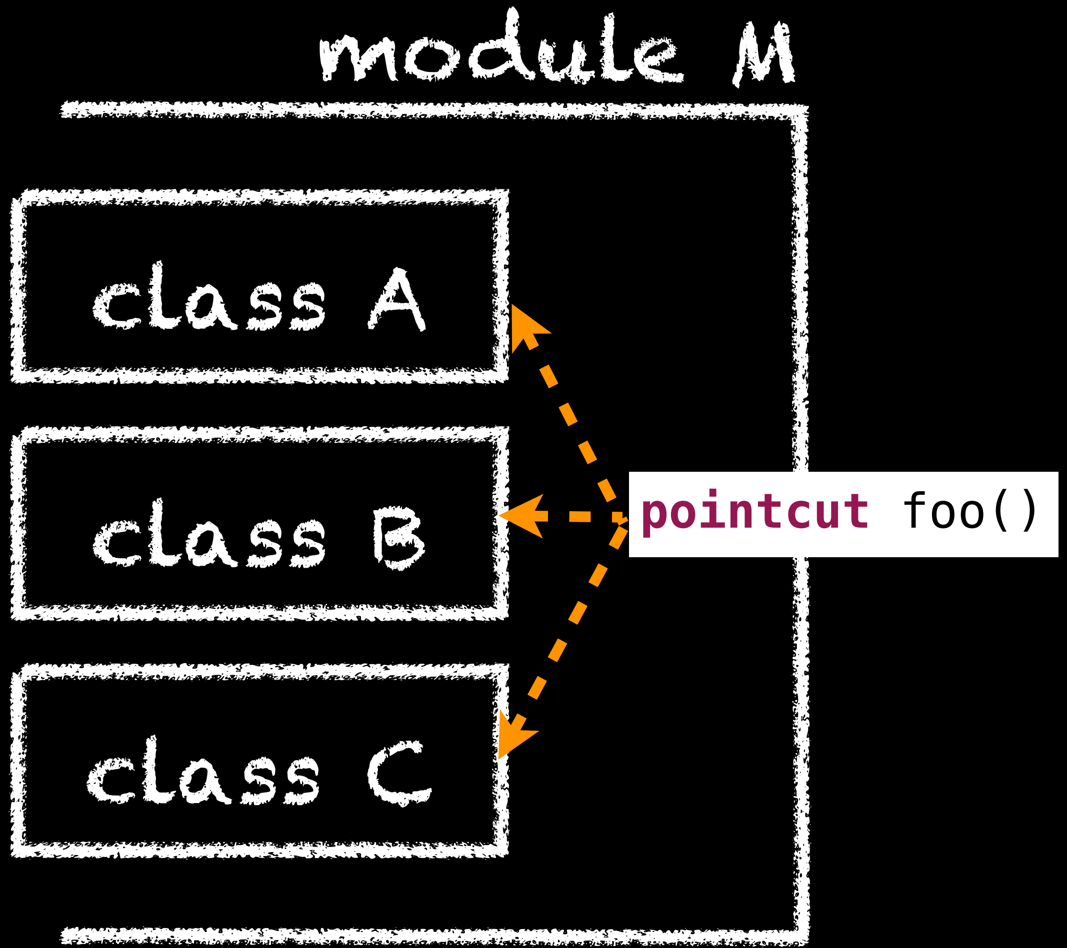
- Open Modules [Aldrich, 2005], etc.



Recovering Modular Reasoning

Putting pointcuts in interfaces [Gudmundson, 2001]

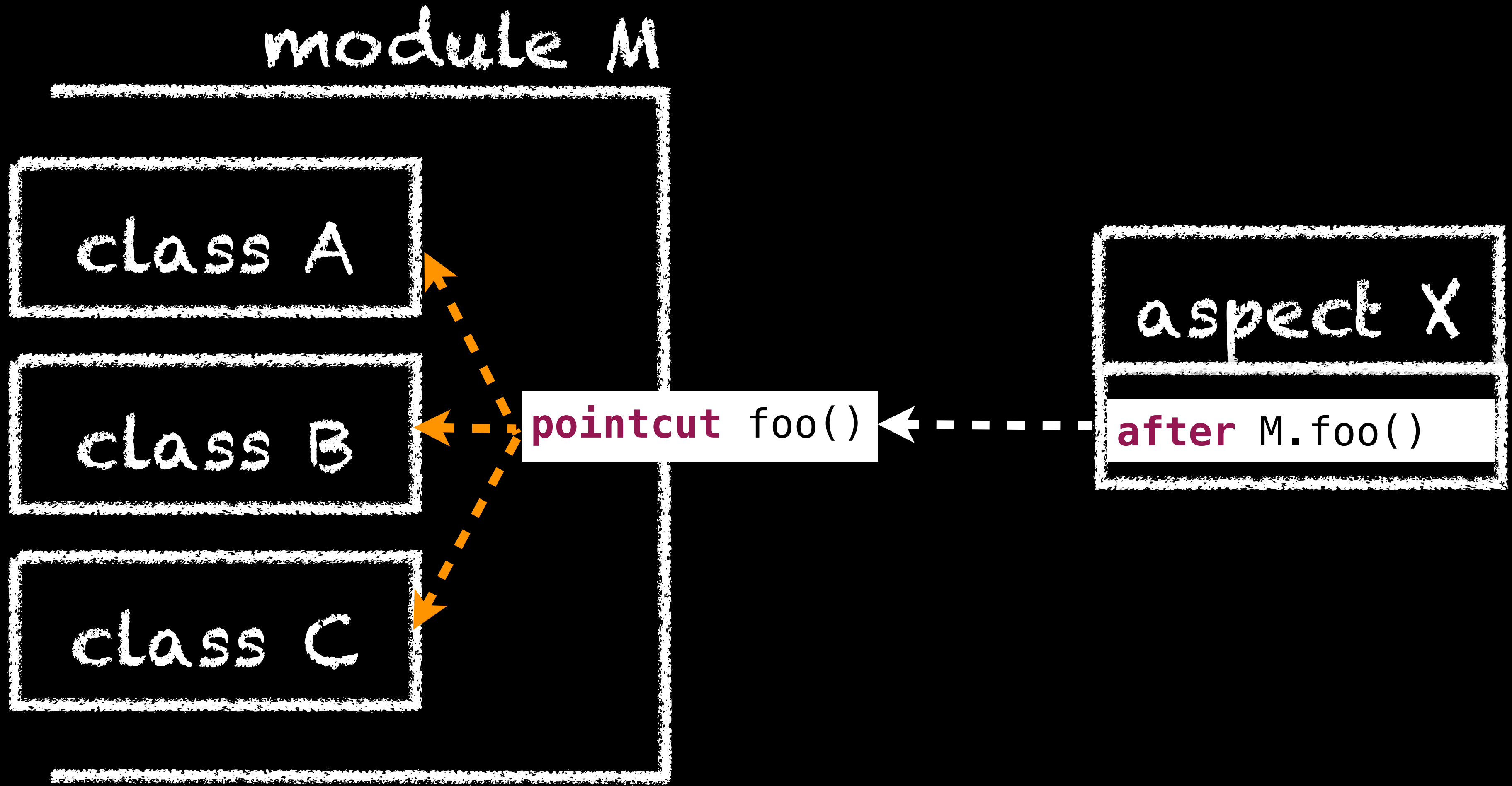
- Open Modules [Aldrich, 2005], etc.



Recovering Modular Reasoning

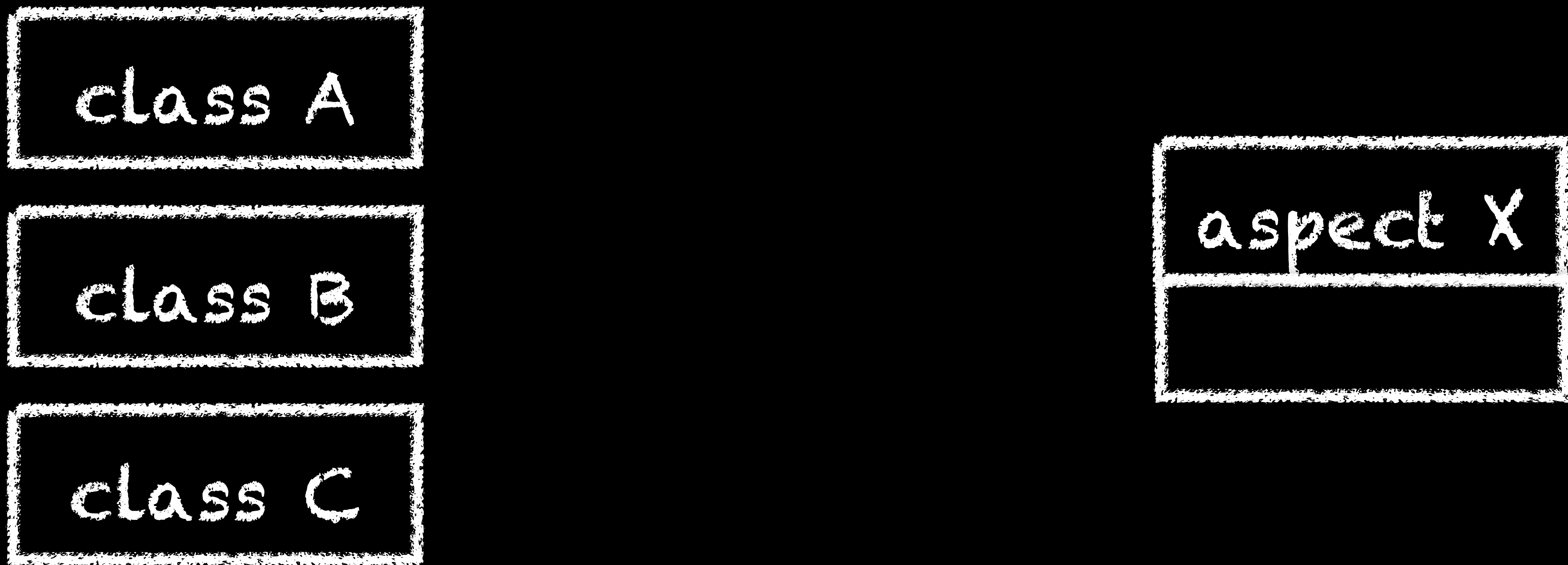
Putting pointcuts in interfaces [Gudmundson, 2001]

- Open Modules [Aldrich, 2005], etc.



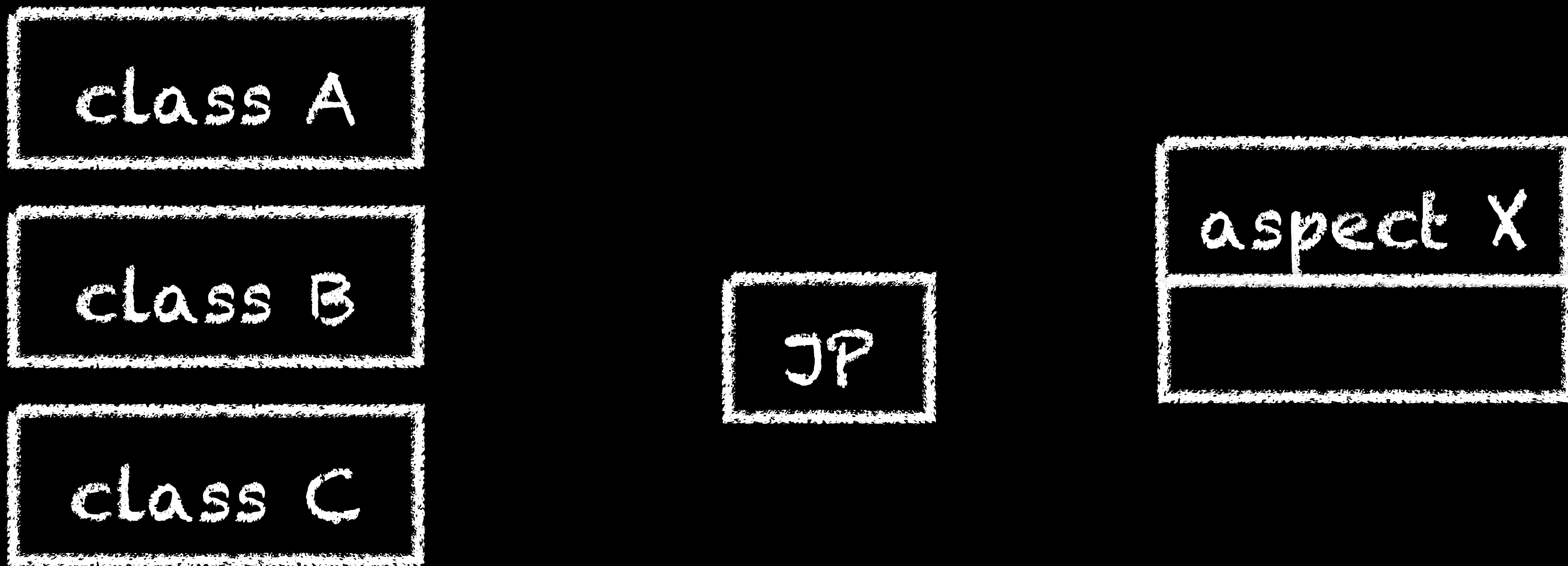
Join Point Types

[Steimann, 2010]



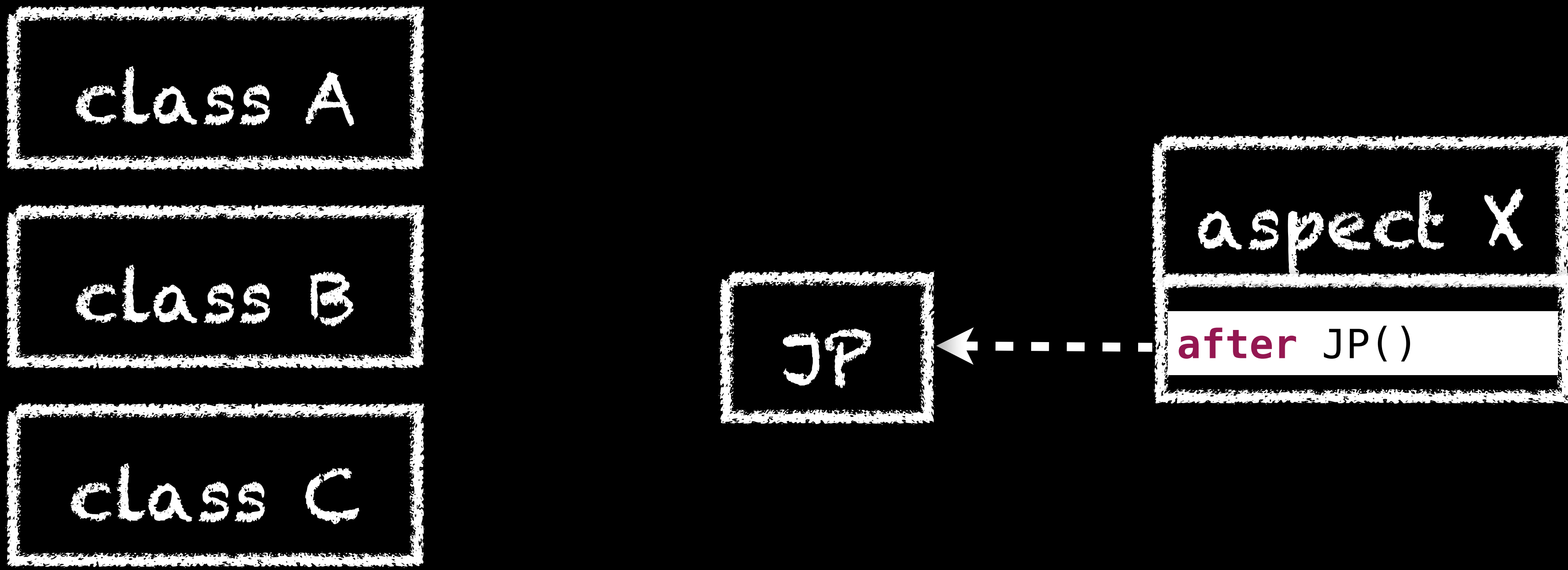
Join Point Types

[Steimann, 2010]



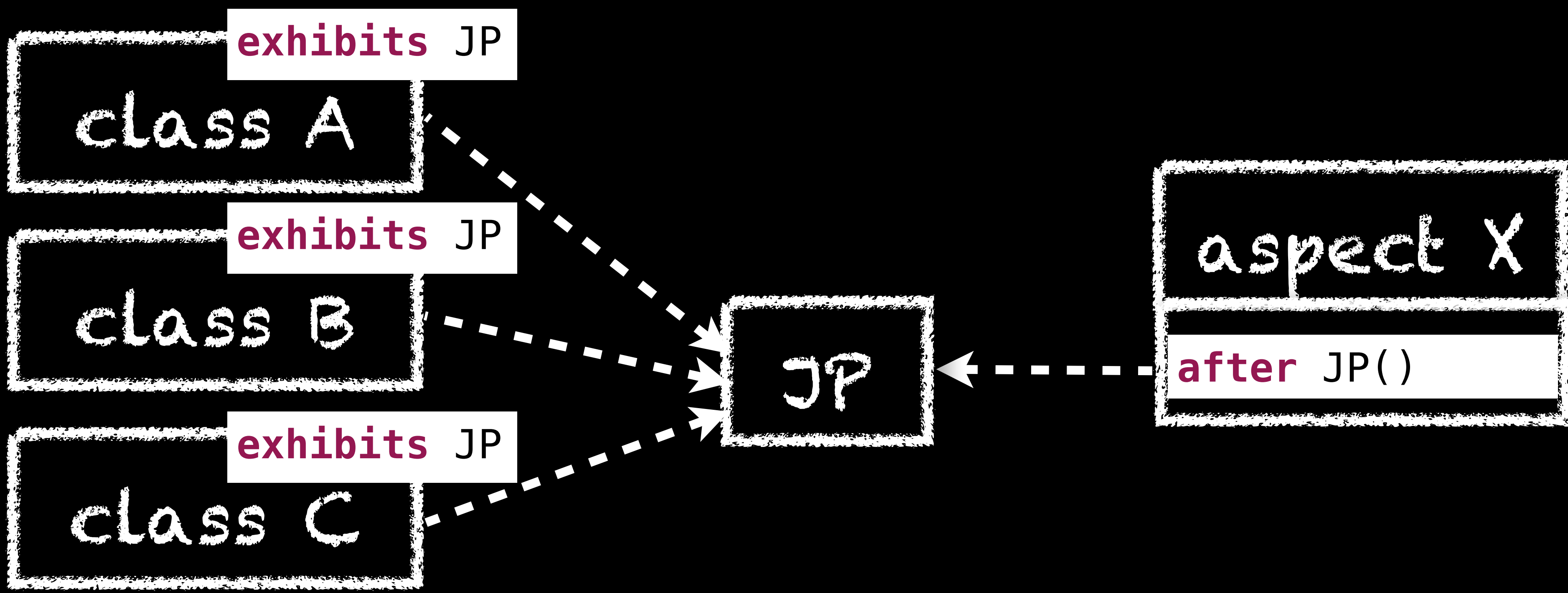
Join Point Types

[Steimann, 2010]



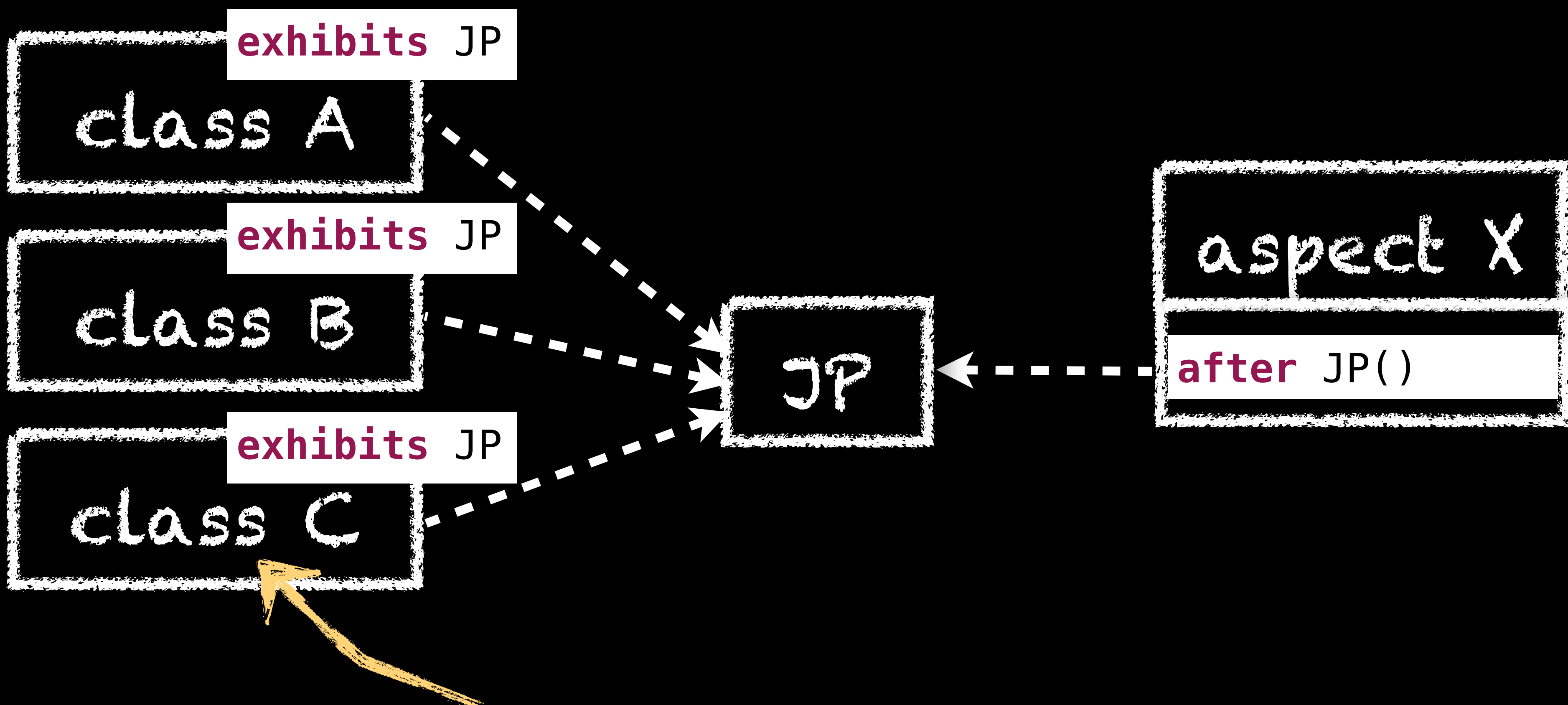
Join Point Types

[Steimann, 2010]



Join Point Types

[Steimann, 2010]



local quantification only!

```
class C exhibits JP {
  pointcut JP: execution(void setX(..)) || ...
  //...
}
```

Modularity Issues

```
joinpointtype CheckingOut {
  float price;
  Customer cus;
}
```

Modularity Issues

```
joinpointtype CheckingOut {
  float price;
  Customer cus;
}
```

```
pointcut CheckingOut(float price, Customer cus);
```

Modularity Issues

```
joinpointtype CheckingOut {
  float price;
  Customer cus;
}
```

same information

```
pointcut CheckingOut(float price, Customer cus);
```

Modularity Issues

```
joinpointtype CheckingOut {
  float price;
  Customer cus;
}
```

same information

```
pointcut CheckingOut(float price, Customer cus);
```

```
interface IFoo {
  m(float p, String s);
}
```

Modularity Issues

```
joinpointtype CheckingOut {
  float price;
  Customer cus;
}
```

same information

```
pointcut CheckingOut(float price, Customer cus);
```

```
interface IFoo {
  ??? m(float p, String s) throws ???;
}
```

return type?

checked exceptions?

Modularity Issues

```
joinpointtype CheckingOut {
  float price;
  Customer cus;
}
```

same information

```
pointcut CheckingOut(float price, Customer cus);
```

```
interface IFoo {
  ??? m(float p, String s) throws ???;
}
```

return type?
checked exceptions?

not type safe

Join Point Interfaces

[Inostroza, 2011] [Bodden, TOSEM]

joint work with
Milton Inostroza
Eric Bodden

Join Point Interfaces

[Inostroza, 2011] [Bodden, TOSEM]

joint work with
Milton Inostroza
Eric Bodden

"Join Point Types Revisited"

Join Point Interfaces

[Inostroza, 2011] [Bodden, TOSEM]

joint work with
Milton Inostroza
Eric Bodden

"Join Point Types Revisited"

- no fragile name dependencies

Join Point Interfaces

joint work with
Milton Inostroza
Eric Bodden

[Inostroza, 2011] [Bodden, TOSEM]

"Join Point Types Revisited"

- no fragile name dependencies
- expressive enough for safe modular type checking

Join Point Interfaces

joint work with
Milton Inostroza
Eric Bodden

[Inostroza, 2011] [Bodden, TOSEM]

"Join Point Types Revisited"

- no fragile name dependencies
- expressive enough for safe modular type checking

```
jpi void CheckingOut(float price, Customer cus) throws IOException
```

Join Point Interfaces

[Inostroza, 2011][Bodden, TOSEM]

joint work with
Milton Inostroza
Eric Bodden

"Join Point Types Revisited"

- no fragile name dependencies
- expressive enough for safe modular type checking

```
jpi void CheckingOut(float price, Customer cus) throws IOException
```

Fix other shortcomings

- join point polymorphism semantics (multiple dispatch)
- unsound use of variant typing (later)
- etc.

Quantification Issues

Quantification Issues

Some aspects are inherently "wide"

- dynamic analyses, system-wide properties, etc.
- require a lot of exhibit clauses

Quantification Issues

Some aspects are inherently “wide”

- dynamic analyses, system-wide properties, etc.
- require a lot of exhibit clauses

Case study

- port existing “Law Of Demeter” checking aspect

Quantification Issues

Some aspects are inherently “wide”

- dynamic analyses, system-wide properties, etc.
- require a lot of exhibit clauses

Case study

- port existing “Law Of Demeter” checking aspect

	# exhibits
LawOfDemeter	130

Quantification Issues

Some aspects are inherently "wide"

- dynamic analyses, system-wide properties, etc.
- require a lot of exhibit clauses

Case study

- port existing "Law Of Demeter" checking aspect

	# **exhibits**
LawOfDemeter	**130**

Cannot really ignore this kind of aspects!

Controlled Global Quantification

```
jpi JP(): execution(* *.*(..))
```

Controlled Global Quantification

```
jpi JP(): execution(* *.*(..))
```

Controlled Global Quantification

```
jpi JP(): execution(* *.*(..))
```

white box

```
class A {
  //...
}
```

Controlled Global Quantification

```
jpi JP(): execution(* *.*(..))
```

white box

```
class A {
  //...
}
```

black box

```
sealed class C {
  //...
}
```

Controlled Global Quantification

```
jpi JP(): execution(* *.*(..))
```

white box

```
class A {
  //...
}
```

black box

```
sealed class C {
  //...
}
```

(can still expose other JPIs)

Controlled Global Quantification

```
jpi JP(): execution(* *.*(..))
```

white box

```
class A {
  //...
}
```

black box

```
sealed class C {
  //...
}
```

(can still expose other JPIs)

grey box

```
class B {
  exhibits JP(): global() && !execution(* secret(..));
  //...
}
```

Controlled Global Quantification

```
jpi JP(): execution(* *.*(..))
```

white box

```
class A {
  //...
}
```

black box

```
sealed class C {
  //...
}
```

(can still expose other JPIs)

grey box

```
class B {
  exhibits JP(): global() && !execution(* secret(..));
  //...
}
```

Perspectives

Perspectives

vs

Perspectives

Modular reasoning

vs

Perspectives

Modular reasoning	vs	Unanticipated extension points

Perspectives

Modular reasoning	vs	Unanticipated extension points

Resolving this tension is crucial

Perspectives

Modular reasoning vs Unanticipated extension points

Resolving this tension is crucial

- look back at work on Open Implementations [Kiczales, 1997]

Perspectives

Modular reasoning

vs

Unanticipated extension points

Resolving this tension is crucial

- look back at work on Open Implementations [Kiczales, 1997]
- exploit a taxonomy of aspects

Perspectives

Modular reasoning vs Unanticipated extension points

Resolving this tension is crucial

- look back at work on Open Implementations [Kiczales, 1997]
- exploit a taxonomy of aspects
    - quantification: narrow vs. wide

Perspectives

Modular reasoning vs Unanticipated extension points

Resolving this tension is crucial

- look back at work on Open Implementations [Kiczales, 1997]
- exploit a taxonomy of aspects
  - quantification: narrow vs. wide
  - life cycle: development vs. production

Scoping

Interfaces

Types

Effects

Scoping

Interfaces

Types

Effects

Can we ensure that aspects do not break type soundness?

Interaction with other features?
(eg. polymorphism)

Typing Aspects

Safe pointcut/advice binding

- advice can replace computation
- should not introduce runtime type errors

Typing Aspects

Safe pointcut/advice binding

- advice can replace computation
- should not introduce runtime type errors

well-typed
base program

well-typed aspect (?)

Typing Aspects

Safe pointcut/advice binding

- advice can replace computation
- should not introduce runtime type errors

well-typed
base program

+

well-typed aspect (?)

→

well-typed
composed program
???

Subtype Polymorphism

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature
  =proceed

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature
  =proceed

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature
  =proceed

```
Person -> void
```

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature
  =proceed

`Person -> void`   `Person -> void`

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature
  =proceed

Person -> void **Person -> void** **Employee -> void**

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature

=proceed

Person -> void **Person -> void** **Employee -> void**

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

```
Integer around(): call(Number *()){
  Integer i = proceed();
  return i;
}
```

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature
  =proceed

`Person -> void` `Person -> void` `Employee -> void`

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

`void -> Number`

```
Integer around(): call(Number *()){
  Integer i = proceed();
  return i;
}
```

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature
  =proceed

Person -> void **Person -> void** **Employee -> void**

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

void -> Integer **void -> Number**

```
Integer around(): call(Number *()){
  Integer i = proceed();
  return i;
}
```

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature
  =proceed

Person -> void **Person -> void** **Employee -> void**

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

void -> Integer **void -> Number** **void -> Number**

```
Integer around(): call(Number *()){
  Integer i = proceed();
  return i;
}
```

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature
  =proceed

Person -> void **Person -> void** **Employee -> void**

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

void -> Integer **void -> Number** **void -> Number**

```
Integer around(): call(Number *()){
  Integer i = proceed();
  return i;
}
```

unsafe!

Aspects and Subtyping

Principles

- body of advice must adhere to advice signature
- pointcut signature <: join point signatures
- advice signature <: pointcut signature
  =proceed

Person -> void **Person -> void** **Employee -> void**

```
void around(Person p): execution(void *()) && this(p){
  proceed(new Person());
}
```

void -> Integer **void -> Number** **void -> Number**

```
Integer around(): call(Number *()){
  Integer i = proceed();
  return i;
}
```

unsafe!
(AspectJ, Join Point Types)

Invariance in Practice

joint work with
Milton Inostroza
Eric Bodden
[Bodden, TOSEM]

Invariance in Practice

joint work with
Milton Inostroza
Eric Bodden
[Bodden, TOSEM]

A simple solution is to prohibit type variance

- first version of JPIs
- is it practical?

Invariance in Practice

joint work with
Milton Inostroza
Eric Bodden
[Bodden, TOSEM]

A simple solution is to prohibit type variance

- first version of JPIs
- is it practical?

Case study

- port AJHotDraw and LawOfDemeter to JPI

Invariance in Practice

joint work with
Milton Inostroza
Eric Bodden
[Bodden, TOSEM]

A simple solution is to prohibit type variance

- first version of JPIs
- is it practical?

Case study

- port AJHotDraw and LawOfDemeter to JPI

advices

	AspectJ	JPI
AJHotDraw	49	**77**
LawOfDemeter	6	**68**

Recovering Flexibility

[Jagadeesan, 2006]

Recovering Flexibility

Generic JPIs

- type parameters [Jagadeesan, 2006]

Recovering Flexibility

Generic JPIs

- type parameters [Jagadeesan, 2006]

```
<R,A,B> jpi R MethodCall(A thiz, B targt);
```

Recovering Flexibility

Generic JPIs

- type parameters [Jagadeesan, 2006]

```
<R,A,B> jpi R MethodCall(A thiz, B targt);
```

advices

	AspectJ	JPI v1	JPI v2
AJHotDraw	49	77	**49**
LawOfDemeter	6	68	**6**

Recovering Flexibility

Generic JPIs

- type parameters [Jagadeesan, 2006]

```
<R,A,B> jpi R MethodCall(A thiz, B targt);
```

advices

	AspectJ	JPI v1	JPI v2
AJHotDraw	49	77	**49**
LawOfDemeter	6	68	**6**

- lose the ability to do replacement advice (parametricity)

Recovering Flexibility

Generic JPIs

- type parameters [Jagadeesan, 2006]

```
<R,A,B> jpi R MethodCall(A thiz, B targt);
```

advices

	AspectJ	JPI v1	JPI v2
AJHotDraw	49	77	**49**
LawOfDemeter	6	68	**6**

- lose the ability to do replacement advice (parametricity)

Beyond genericity: type ranges [De Fraine, 2008/2010]

- flexible type-safe replacement advice
- ... added complexity (no free lunch :/)

Parametric Polymorphism

joint work with

Ismael Figueroa

Nicolas Tabareau

A Typed Functional Embedding of First-Class Aspects

A Typed Functional Embedding of First-Class Aspects

join points represent function applications

```
fib 10
```

A Typed Functional Embedding of First-Class Aspects

join points represent function applications

```
fib 10
```

```
data JP a b = JP (a → b) a
```

A Typed Functional Embedding of First-Class Aspects

join points represent function applications

```
fib 10
```

```
data JP a b = JP (a → b) a
```

a pointcut is a predicate on any join point

```
pcCall fib
```

A Typed Functional Embedding of First-Class Aspects

join points represent function applications

```
fib 10
```

```
data JP a b = JP (a → b) a
```

a pointcut is a predicate on any join point

```
pcCall fib
```

```
data PC = PC (forall a b. JP a b → Bool)
```

A Typed Functional Embedding of First-Class Aspects

join points represent function applications

```
fib 10
```

```
data JP a b = JP (a → b) a
```

a pointcut is a predicate on any join point

```
pcCall fib
```

```
data PC = PC (forall a b. JP a b → Bool)
```

an advice is a function transformer

```
memoize proceed n = ...
```

A Typed Functional Embedding of First-Class Aspects

join points represent function applications

```
fib 10
```

```
data JP a b = JP (a → b) a
```

a pointcut is a predicate on any join point

```
pcCall fib
```

```
data PC = PC (forall a b. JP a b → Bool)
```

an advice is a function transformer

```
memoize proceed n = ...
```

```
type Advice a b = (a → b) → a → b
```

A Typed Functional Embedding of First-Class Aspects

join points represent function applications

```
fib 10
```

```
data JP a b = JP (a → b) a
```

a pointcut is a predicate on any join point

```
pcCall fib
```

```
data PC = PC (forall a b. JP a b → Bool)
```

an advice is a function transformer

```
memoize proceed n = ...
```

```
type Advice a b = (a → b) → a → b
```

an aspect is a pc/adv binding

```
aspect (pcCall fib) memoize
```

A Typed Functional Embedding of First-Class Aspects

join points represent function applications

```
fib 10
```

```
data JP a b = JP (a → b) a
```

a pointcut is a predicate on any join point

```
pcCall fib
```

```
data PC = PC (forall a b. JP a b → Bool)
```

an advice is a function transformer

```
memoize proceed n = ...
```

```
type Advice a b = (a → b) → a → b
```

an aspect is a pc/adv binding

```
aspect (pcCall fib) memoize
```

```
data Aspect ... = Aspect PC (Advice a b)
```

A Typed Functional Embedding of First-Class Aspects

join points represent function applications

```
fib 10
```

```
data JP a b = JP (a → b) a
```

a pointcut is a predicate on any join point

```
pcCall fib
```

```
data PC = PC (forall a b. JP a b → Bool)
```

an advice is a function transformer

```
memoize proceed n = ...
```

```
type Advice a b = (a → b) → a → b
```

an aspect is a pc/adv binding

```
aspect (pcCall fib) memoize
```

```
data Aspect ... = Aspect PC (Advice a b)
```

how to ensure the aspect is well-typed?

A Typed Functional Embedding of First-Class Aspects

join points represent function applications

```
fib 10
```

```
data JP a b = JP (a → b) a
```

a pointcut is a predicate on any join point

```
pcCall fib
```

```
data PC = PC (forall a b. JP a b → Bool)
```

an advice is a function transformer

```
memoize proceed n = ...
```

```
type Advice a b = (a → b) → a → b
```

an aspect is a pc/adv binding

```
aspect (pcCall fib) memoize
```

```
data Aspect ... = Aspect PC (Advice a b)
```

how to ensure the aspect is well-typed?

(broken)

Matched Types

annotate PC with their **matched type**

```
data PC a b = PC (forall a’ b’. JP a’ b’ → Bool)
```

Matched Types

annotate PC with their **matched type**

```
data PC a b = PC (forall a' b'. JP a' b' → Bool)
```

possibly matches applications of functions a→b

Matched Types

annotate PC with their **matched type**

```
data PC a b = PC (forall a’ b’. JP a’ b’ → Bool)
```

possibly matches applications of functions a→b

```
pc = pcCall id
```

Matched Types

annotate PC with their **matched type**

```
data PC a b = PC (forall a' b'. JP a' b' → Bool)
```

possibly matches applications of functions a→b

```
pc :: PC a a
pc = pcCall id
```

Matched Types

annotate PC with their **matched type**

```
data PC a b = PC (forall a' b'. JP a' b' → Bool)
```

possibly matches applications of functions a→b

```
pc :: PC a a
pc = pcCall id
```

enforce that both types are compatible

```
data Aspect a b = Aspect (PC a b) (Advice a b)
```

Matched Types

annotate PC with their **matched type**

```
data PC a b = PC (forall a' b'. JP a' b' → Bool)
```

possibly matches applications of functions a→b

```
pc :: PC a a
pc = pcCall id
```

enforce that both types are compatible

```
data Aspect a b = Aspect (PC a b) (Advice a b)
```

Matched Types

annotate PC with their **matched type**

```
data PC a b = PC (forall a’ b’. JP a’ b’ → Bool)
```

possibly matches applications of functions a→b

```
pc :: PC a a
pc = pcCall id
```

enforce that both types are compatible

```
data Aspect a b = Aspect (PC a b) (Advice a b)
```

(broken)

```haskell
data Aspect a b = Aspect (PC a b) (Advice a b)
```

```haskell
data Aspect a b = Aspect (PC a b) (Advice a b)
```

```haskell
pc :: PC a  a
pc = pcCall id
```

```haskell
data Aspect a b = Aspect (PC a b) (Advice a b)
```

```haskell
pc :: PC a  a
pc = pcCall id
```

```haskell
adv :: Advice Char Char
adv proceed c = proceed (toUpper c)
```

```haskell
data Aspect a b = Aspect (PC a b) (Advice a b)
```

```haskell
pc :: PC a  a
pc = pcCall id
```

```haskell
adv :: Advice Char Char
adv proceed c = proceed (toUpper c)
```

```haskell
data Aspect a b = Aspect (PC a b) (Advice a b)
```

```haskell
pc :: PC a  a
pc = pcCall id
```

```haskell
adv :: Advice Char Char
adv proceed c = proceed (toUpper c)
```

✓ unifiable

```
data Aspect a b = Aspect (PC a b) (Advice a b)
```

```
pc :: PC a  a
pc = pcCall id
```

```
adv :: Advice Char Char
adv proceed c = proceed (toUpper c)
```

unifiable

```
id ‘a’
```

```haskell
data Aspect a b = Aspect (PC a b) (Advice a b)
```

```haskell
pc :: PC a  a
pc = pcCall id
```

```haskell
adv :: Advice Char Char
adv proceed c = proceed (toUpper c)
```

✔ unifiable

```haskell
id ‘a’
```

✔

```haskell
id [True,False]
```

✘

```
data Aspect a b = Aspect (PC a b) (Advice a b)
```

```
pc :: PC a  a
pc = pcCall id
```

```
adv :: Advice Char Char
adv proceed c = proceed (toUpper c)
```

unifiable

```
id ‘a’
```

```
id [True,False]
```

Problem: **unification is symmetric**

Well-typed Aspects

```
data Aspect a b c d = Aspect (PC a b) (Advice c d)
```

Well-typed Aspects

```
data Aspect a b c d = Aspect (PC a b) (Advice c d)
```

need to ensure that the matched type a→b
is **less general** than the type of the advice c→d

Well-typed Aspects

```
data Aspect a b c d = Aspect (PC a b) (Advice c d)
```

need to ensure that the matched type a→b
is **less general** than the type of the advice c→d

A multi-parameter type class defines a **relation** between types

Well-typed Aspects

```
data Aspect a b c d = Aspect (PC a b) (Advice c d)
```

need to ensure that the matched type a→b
is **less general** than the type of the advice c→d

A multi-parameter type class defines a **relation** between types

```
data Aspect a b c d = (LessGen (a→b) (c→d)) ⇒
                          Aspect (PC a b) (Advice c d)
```

Well-typed Aspects

```
data Aspect a b c d = Aspect (PC a b) (Advice c d)
```

need to ensure that the matched type a→b
is **less general** than the type of the advice c→d

A multi-parameter type class defines a **relation** between types

```
data Aspect a b c d = (LessGen (a→b) (c→d)) ⟹
                      Aspect (PC a b) (Advice c d)
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```
pcAnd :: PC a b → PC c d → PC e f
```

Composing Pointcuts

```haskell
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```haskell
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```haskell
pcAnd :: PC a b → PC c d → PC e f
```

```haskell
pc1 :: PC Int Int
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```
pcAnd :: PC a b → PC c d → PC e f
```

```
pc1 :: PC Int Int
```

```
pc2 :: PC a a
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```
pcAnd :: PC a b → PC c d → PC e f
```

```
pc1 :: PC Int Int
```

```
pc2 :: PC a a
```

```
:: PC Int Int
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```
pcAnd :: PC a b → PC a b → PC a b
```

```
pc1 :: PC Int Int
```

```
pc2 :: PC a a
```

```
:: PC Int Int
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```
pcAnd :: PC a b → PC a b → PC a b
```

```
pc1 :: PC Int Int
```

```
pcNot :: PC a b → PC c d
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```
pcAnd :: PC a b → PC a b → PC a b
```

```
pc1 :: PC Int Int
```

```
pcNot :: PC a b → PC c d
```

```
pcOr  :: PC a b → PC c d → PC e f
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```
pcAnd :: PC a b → PC a b → PC a b
```

```
pcNot :: PC a b → PC c d
```

```
pcOr  :: PC a b → PC c d → PC e f
```

```
pc1 :: PC Int Int
```

```
pc2 :: PC Int Bool
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```
pcAnd :: PC a b → PC a b → PC a b
```

```
pcNot :: PC a b → PC c d
```

```
pcOr  :: PC a b → PC c d → PC e f
```

```
pc1 :: PC Int Int
```

```
pc2 :: PC Int Bool
```

```
:: PC Int a
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```
pcAnd :: PC a b → PC a b → PC a b
```

```
pcNot :: PC a b → PC c d
```

```
pcOr  :: (LeastGen (a→b) (c→d) (e→f)) ⇒
         PC a b → PC c d → PC e f
```

```
pc1 :: PC Int Int
```

```
pc2 :: PC Int Bool
```

```
:: PC Int a
```

Composing Pointcuts

```
data PC a b = PC (forall a b. JP a b → Bool)
```

how do we get the matched type?

primitive pointcut designators

```
pcCall, pcType :: (a → b) → PC a b
```

logical combinators

```
pcAnd :: PC a b → PC a b → PC a b
```

```
pcNot :: PC a b → PC c d
```

```
pcOr  :: (LeastGen (a→b) (c→d) (e→f)) ⇒
         PC a b → PC c d → PC e f
```

rely on anti-unification

Advantages of the Approach

[Tabareau, 2013]

Advantages of the Approach

[Tabareau, 2013]

Type soundness

- proof follows from correctness of LeastGen
- much simpler than AspectML (ad hoc calculus & type system)

Advantages of the Approach

[Tabareau, 2013]

Type soundness

- proof follows from correctness of LeastGen
- much simpler than AspectML (ad hoc calculus & type system)

More expressive

- first-class advice, extensible set of pointcut designators, bounded polymorphism (type classes)

Advantages of the Approach

[Tabareau, 2013]

Type soundness

- proof follows from correctness of LeastGen
- much simpler than AspectML (ad hoc calculus & type system)

More expressive

- first-class advice, extensible set of pointcut designators, bounded polymorphism (type classes)

Compact implementation

- 1K vs. 15-25K for AspectML and AspectualCaml

Advantages of the Approach

[Tabareau, 2013]

Type soundness

- proof follows from correctness of LeastGen
- much simpler than AspectML (ad hoc calculus & type system)

More expressive

- first-class advice, extensible set of pointcut designators, bounded polymorphism (type classes)

Compact implementation

- 1K vs. 15-25K for AspectML and AspectualCaml

Monadic embedding as a Haskell library

Scoping

Interfaces

Types

Effects

Scoping

Interfaces

Types

Effects

Can we control what advice can do?
(proceed, args/return, side effects)

Beyond Types

Beyond Types

Type soundness does not tell much

- control effects through proceed?
- arbitrary effects?

Beyond Types

Type soundness does not tell much

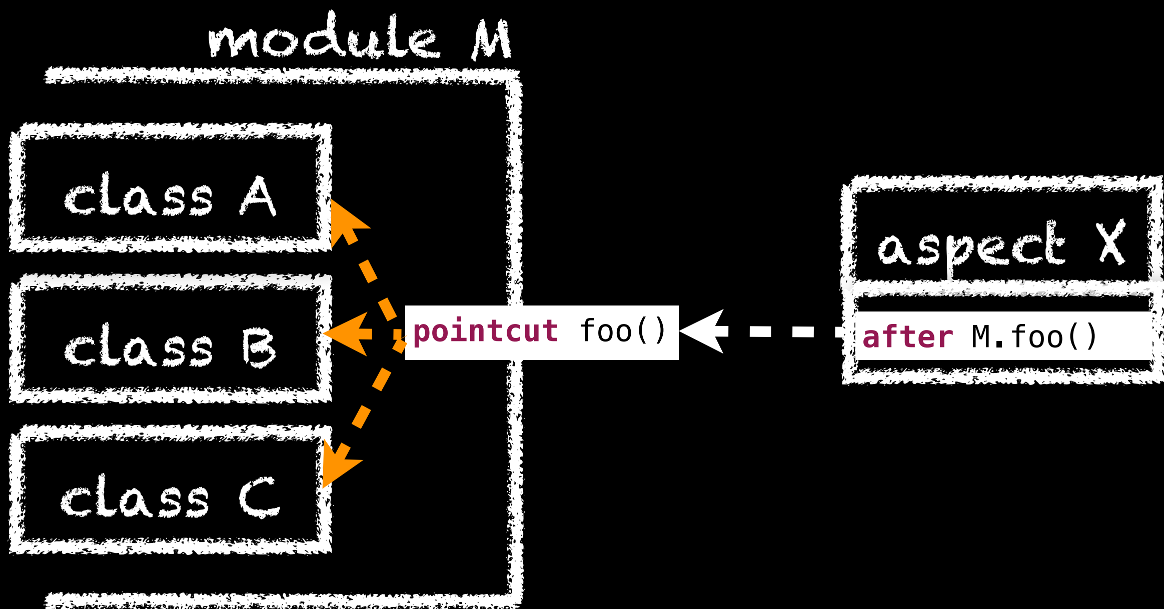
- control effects through proceed?
- arbitrary effects?

Expressive aspect specifications

- black-box behavioral contracts [Skotiniotis, 2004; Zhao, 2003]...
- control effects [Rinard, 2004]
- translucid contracts [Bagherzadeh, 2011]
- model checking [Katz, 2003; Krishnamurthi, 2004]...

Rich Types

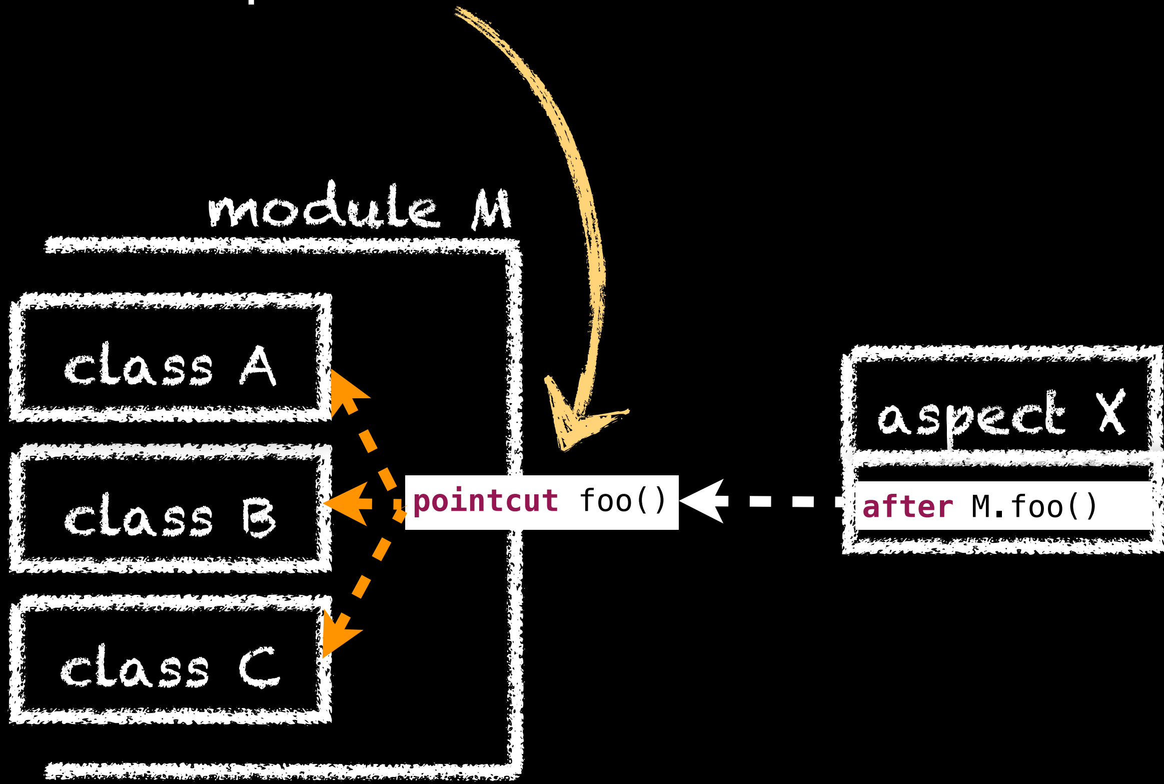
Rich Types



module M
class A
class B
class C
pointcut foo()
aspect X
after M.foo()

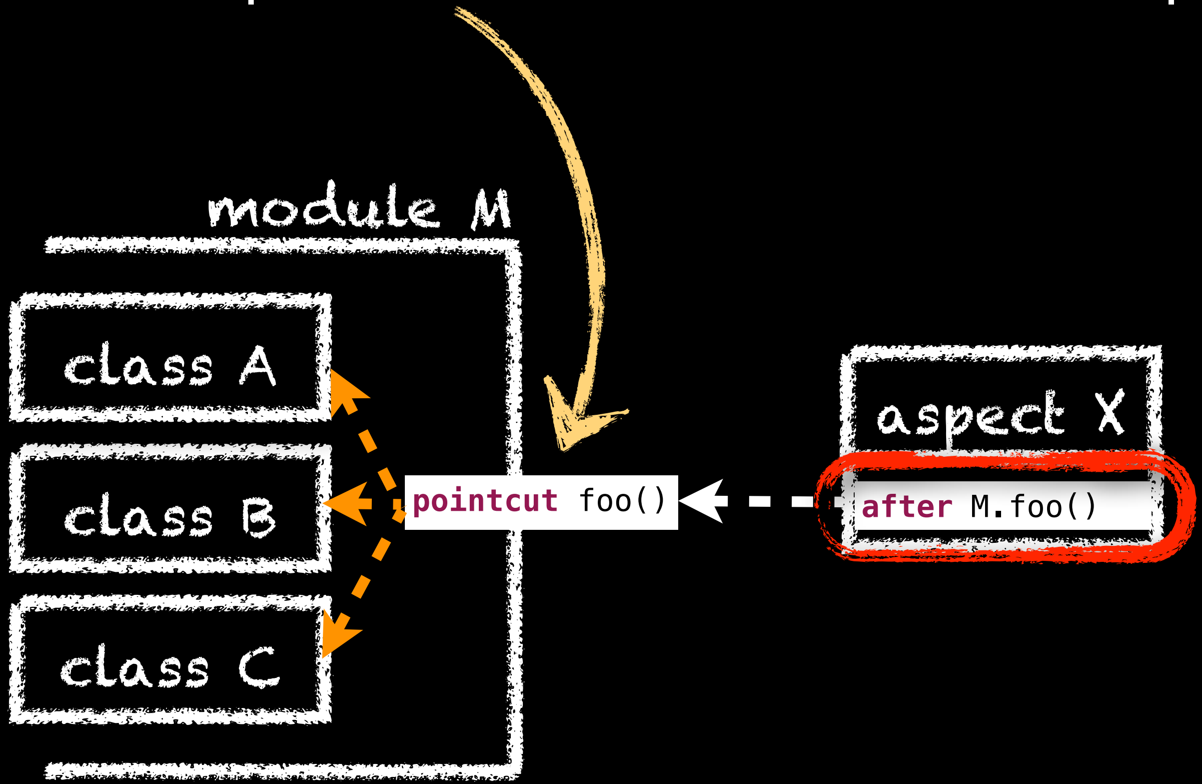
Rich Types

Can we enrich aspect interfaces with effect specs?



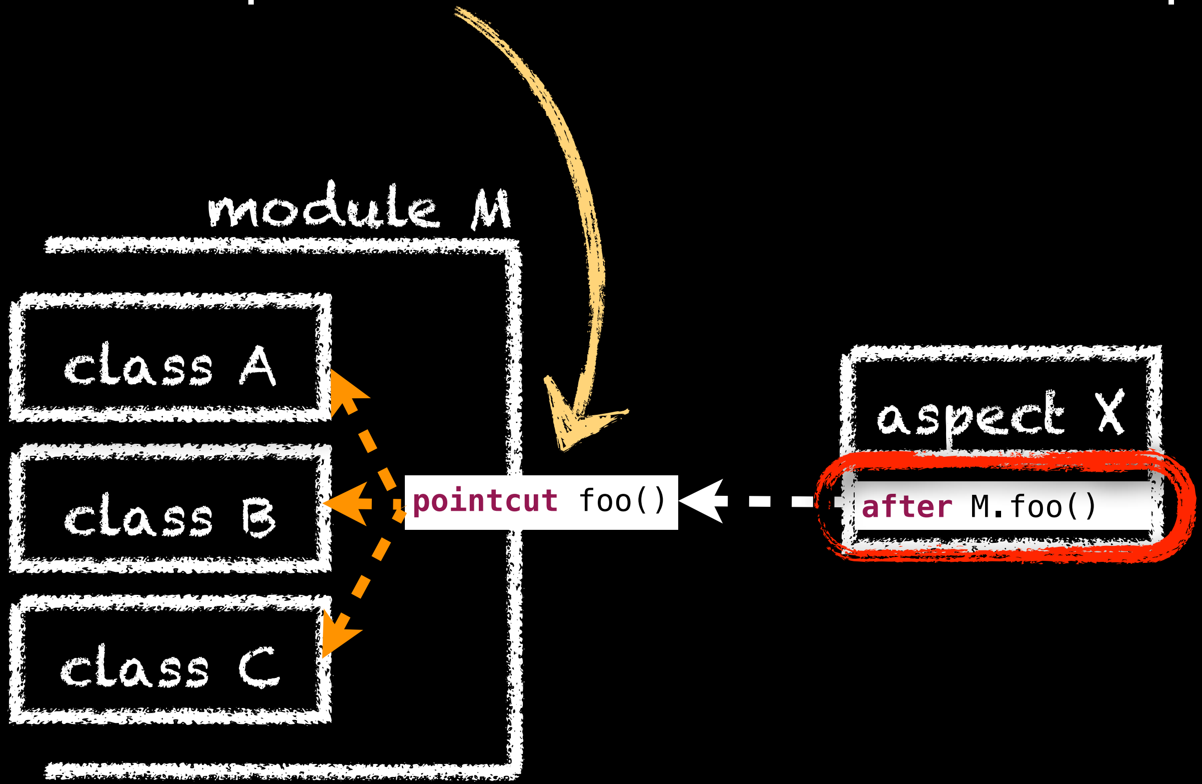
Rich Types

Can we enrich aspect interfaces with effect specs?



Rich Types

Can we enrich aspect interfaces with effect specs?



The Haskell type system deals with effects!

All You Need to Know About Monads (for this talk)

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

monads

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

Several effects

```
foo :: Int → App Int
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

Several effects

```
foo :: Int → App Int
```

```
type App = ReaderT AppConf (StateT AppState IO)
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

Several effects

```
foo :: Int → App Int
```

```
type App = ReaderT AppConf (StateT AppState IO)
```

monad transformers

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

Several effects

```
foo :: Int → App Int
```

```
type App = ReaderT AppConf (StateT AppState IO)
```

All You Need to Know About Monads (for this talk)

Purity is the default

```haskell
foo :: Int → Int
```

Side effects reflected in types

```haskell
foo :: Int → IO Int
```

```haskell
foo :: Int → (State Char) Int
```

Several effects

"effect stack"

```haskell
foo :: Int → App Int
```

```haskell
type App = ReaderT AppConf (StateT AppState IO)
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

IO

"effect stack"

Several effects

```
foo :: Int → App Int
```

```
type App = ReaderT AppConf (StateT AppState IO)
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

“effect stack”

Several effects

```
foo :: Int → App Int
```

```
type App = ReaderT AppConf (StateT AppState IO)
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

Reader AppConf

State AppState

IO

“effect stack”

Several effects

```
foo :: Int → App Int
```

```
type App = ReaderT AppConf (StateT AppState IO)
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

Several effects

```
foo :: Int → App Int
```

```
type App = ReaderT AppConf (StateT AppState IO)
```

Reader AppConf
State AppState
IO

"effect stack"

```
foo :: (Monad m) ⇒ Int → m Int
```

All You Need to Know About Monads (for this talk)

Purity is the default

```
foo :: Int → Int
```

Side effects reflected in types

```
foo :: Int → IO Int
```

```
foo :: Int → (State Char) Int
```

Several effects

```
foo :: Int → App Int
```

```
type App = ReaderT AppConf (StateT AppState IO)
```

Reader AppConf
State AppState
IO

“effect stack”

```
foo :: (Monad m) ⇒ Int → m Int
```

(we’ll omit the constraints on monadic type variables)

Talking About Effects

joint work with
Ismael Figueroa
Nicolas Tabareau
[Tabareau, 2013]

Parametrize the model by the effect stack

```
data JP a b = JP (a → b) a
data PC a b = PC (forall a’ b’. JP a’ b’ → Bool)
type Advice a b = (a → b) → a → b
data Aspect a b c d =
  (LessGen (a→b) (c→d)) ⇒ Aspect (PC a b) (Advice c d)
```

Talking About Effects

joint work with
Ismael Figueroa
Nicolas Tabareau
[Tabareau, 2013]

Parametrize the model by the effect stack

```
data JP m a b = JP (a → m b) a
data PC m a b = PC (forall a’ b’. m JP a’ b’ → m Bool)
type Advice m a b = (a → m b) → a → m b
data Aspect m a b c d =
  (LessGen (a→b) (c→d)) ⟹ Aspect (PC m a b) (Advice m c d)
```

Talking About Effects

joint work with
Ismael Figueroa
Nicolas Tabareau
[Tabareau, 2013]

Parametrize the model by the effect stack

```
data JP m a b = JP (a → m b) a
data PC m a b = PC (forall a’ b’. m JP a’ b’ → m Bool)
type Advice m a b = (a → m b) → a → m b
data Aspect m a b c d =
  (LessGen (a→b) (c→d)) ⟹ Aspect (PC m a b) (Advice m c d)
```

Computation happens within the AOT monad transformer

```
newtype AOT m a = ...
```

(used to pass the aspect environment around)

Embedding Type Constraints in Pointcuts

Embedding Type Constraints in Pointcuts

```
module Fib (fib, pcFib) where
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, pcFib) where
```

```
innerFib = ...
fib = ...
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, pcFib) where
```

```
innerFib = ...
fib = ...
```

```
pcFib :: PC m Int Int
pcFib = pcCall innerFib
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, pcFib) where
```

```
innerFib = ...
fib = ...
```

```
pcFib :: PC m Int Int
pcFib = pcCall innerFib
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, pcFib) where
```

```
innerFib = ...
fib = ...
```

```
pcFib :: PC m Int Int
pcFib = pcCall innerFib
```

```
comb :: t → Advice m a b
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, pcFib) where
```

```
innerFib = ...
fib = ...
```

```
pcFib :: PC m Int Int
pcFib = pcCall innerFib
```

```
comb :: t → Advice m a b
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, ppcFib) where
```

```
innerFib = ...
fib = ...
```

```
ppcFib :: ProtectedPC m Int Int t a b
ppcFib = protectPC pcFib comb
```

```
pcFib :: PC m Int Int
pcFib = pcCall innerFib
```

```
comb :: t → Advice m a b
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, ppcFib) where
```

```
innerFib = ...
fib = ...
```

```
ppcFib :: ProtectedPC m Int Int t a b
ppcFib = protectPC pcFib comb
```

```
myadvice :: t
myadvice = ...
```

```
pcFib :: PC m Int Int
pcFib = pcCall innerFib
```

```
comb :: t → Advice m a b
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, ppcFib) where
```

```
innerFib = ...
fib = ...
```

```
ppcFib :: ProtectedPC m Int Int t a b
ppcFib = protectPC pcFib comb
```

```
myadvice :: t
myadvice = ...
```

```
pcFib :: PC m Int Int
pcFib = pcCall innerFib
```

```
myaspect = aspect ppcFib myadvice
```

```
comb :: t → Advice m a b
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, ppcFib) where
```

```
innerFib = ...
fib = ...
```

```
ppcFib :: ProtectedPC m Int Int t a b
ppcFib = protectPC pcFib comb
```

```
myadvice :: t
myadvice = ...
```

```
pcFib :: PC m Int Int
pcFib = pcCall innerFib
```

```
myaspect = aspect ppcFib myadvice
```

```
comb :: t → Advice m a b
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, ppcFib) where
```

```
innerFib = ...
fib = ...
```

```
ppcFib :: ProtectedPC m Int Int t a b
ppcFib = protectPC pcFib comb
```

```
myadvice :: t
myadvice = ...
```

```
pcFib :: PC m Int Int
pcFib = pcCall innerFib
```

```
myaspect = aspect ppcFib myadvice
```

```
comb :: t → Advice m a b
```

Embedding Type Constraints in Pointcuts

```
module Fib (fib, ppcFib) where
```

```
innerFib = ...
fib = ...
```

```
ppcFib :: ProtectedPC m Int Int t a b
ppcFib = protectPC pcFib comb
```

```
myadvice :: t
myadvice = ...
```

```
pcFib :: PC m Int Int
pcFib = pcCall innerFib
```

```
comb :: t → Advice m a b
```

```
myaspect = aspect ppcFib myadvice
```

control effects & side effects

Control Flow Effects

[Rinard, 2004]

Control Flow Effects

[Rinard, 2004]	**definition**
combination	free
replacement	no proceed
augmentation	proceed once same arg/ret
narrowing	proceed at most once same arg/ret

Control Flow Effects

EffectiveAdvice
[Oliveira, 2010]

[Rinard, 2004]	definition	type
combination	free	`Advice m a b`
replacement	no proceed	
augmentation	proceed once same arg/ret	
narrowing	proceed at most once same arg/ret	

Control Flow Effects

EffectiveAdvice
[Oliveira, 2010]

[Rinard, 2004]	definition	type
combination	free	`Advice m a b`
replacement	no proceed	no access to proceed `Replace m a b`
augmentation	proceed once same arg/ret	
narrowing	proceed at most once same arg/ret	

Wednesday, April 17, 13

Control Flow Effects

EffectiveAdvice
[Oliveira, 2010]

[Rinard, 2004]	definition	type
combination	free	`Advice m a b`
replacement	no proceed	no access to proceed `Replace m a b`
augmentation	proceed once same arg/ret	pair before/after `Augment m a b c`
narrowing	proceed at most once same arg/ret	

Control Flow Effects

EffectiveAdvice
[Oliveira, 2010]

[Rinard, 2004]	definition	type
combination	free	`Advice m a b`
replacement	no proceed	no access to proceed `Replace m a b`
augmentation	proceed once same arg/ret	pair before/after `Augment m a b c`
narrowing	proceed at most once same arg/ret	predicate + rep + aug `Narrow m a b c`

Control Flow Effects

EffectiveAdvice
[Oliveira, 2010]

[Rinard, 2004]	definition	type
combination	free	`Advice m a b`
replacement	no proceed	no access to proceed `Replace m a b`
augmentation	proceed once same arg/ret	pair before/after `Augment m a b c`
narrowing	proceed at most once same arg/ret	predicate + rep + aug `Narrow m a b c`

memoization?

Control Flow Effects

EffectiveAdvice
[Oliveira, 2010]

[Rinard, 2004]	definition	type
combination	free	`Advice m a b`
replacement	no proceed	no access to proceed `Replace m a b`
augmentation	proceed once same arg/ret	pair before/after `Augment m a b c`
narrowing	proceed at most once same arg/ret	predicate + rep + aug `Narrow m a b c`

memoization?

Enforcing Narrowing Advice

```
type Narrow m a b c = (a → m Bool, Augment m a b c, Replace m a b)
```

Enforcing Narrowing Advice

```
type Narrow m a b c = (a → m Bool, Augment m a b c, Replace m a b)
```

combinator that requires Narrow

```
narrow :: Narrow m a b c → Advice m a b
```

Enforcing Narrowing Advice

```
type Narrow m a b c = (a → m Bool, Augment m a b c, Replace m a b)
```

combinator that requires Narrow

```
narrow :: Narrow m a b c → Advice m a b

narrow (pred, aug, rep) proceed x =
  do b <- pred x
     if b then replace rep proceed x
          else augment aug proceed x
```

Enforcing Narrowing Advice

```
type Narrow m a b c = (a → m Bool, Augment m a b c, Replace m a b)
```

combinator that requires Narrow

```
narrow :: Narrow m a b c → Advice m a b
```

Enforcing Narrowing Advice

```
type Narrow m a b c = (a → m Bool, Augment m a b c, Replace m a b)
```

combinator that requires Narrow

```
narrow :: Narrow m a b c → Advice m a b
```

```
module Fib (fib, ppcFib) where
```

```
ppcFib = protectPC pcFib narrow
```

Enforcing Narrowing Advice

```
type Narrow m a b c = (a → m Bool, Augment m a b c, Replace m a b)
```

combinator that requires Narrow

```
narrow :: Narrow m a b c → Advice m a b
```

```
module Fib (fib, ppcFib) where
```

```
ppcFib = protectPC pcFib narrow
```

Enforcing Narrowing Advice

```
type Narrow m a b c = (a → m Bool, Augment m a b c, Replace m a b)
```

combinator that requires Narrow

```
narrow :: Narrow m a b c → Advice m a b
```

```
module Fib (fib, ppcFib) where
```

```
ppcFib = protectPC pcFib narrow
```

```
memoize :: Narrow ...
memoize = ...
```

Enforcing Narrowing Advice

```
type Narrow m a b c = (a → m Bool, Augment m a b c, Replace m a b)
```

combinator that requires Narrow

```
narrow :: Narrow m a b c → Advice m a b
```

```
module Fib (fib, ppcFib) where
```

```
ppcFib = protectPC pcFib narrow
```

```
memoize :: Narrow ...
memoize = ...
```

```
crazy :: Advice ...
crazy = ...
```

Effect Interference

Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]

Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]

```
AOT m a
```

Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]

```
AOT m a
```

Error d
Reader c
State b
IO

Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]

`NIAOT t m a`

Error d

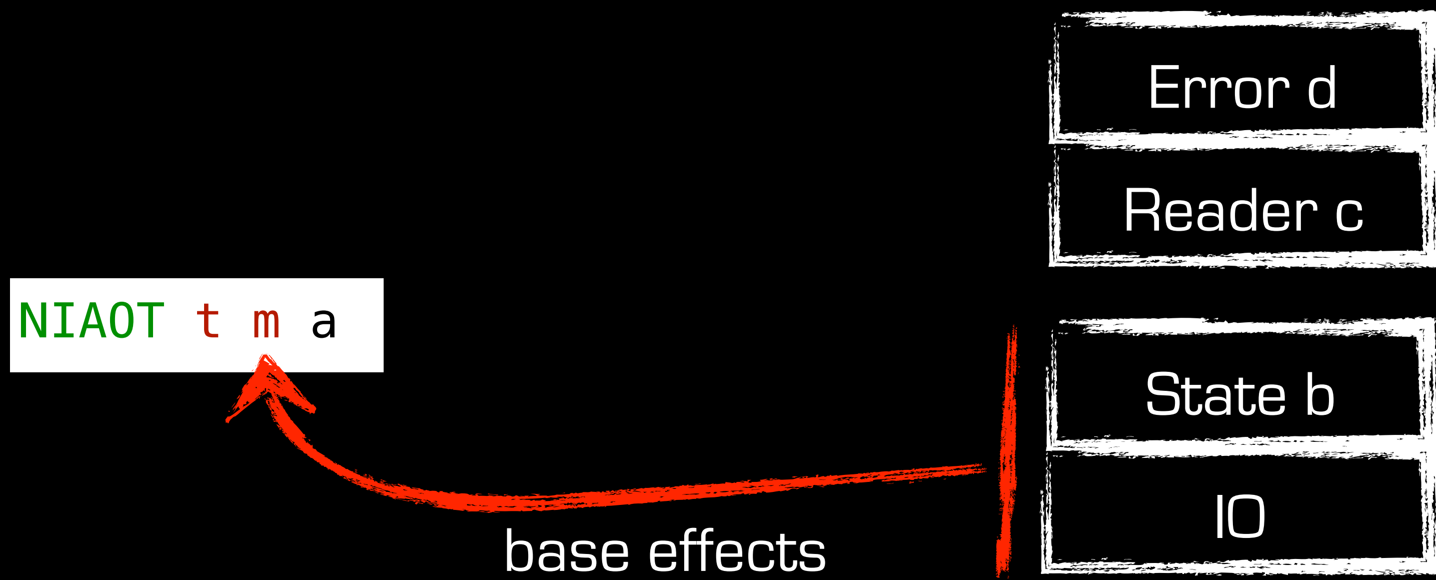
Reader c

State b

IO

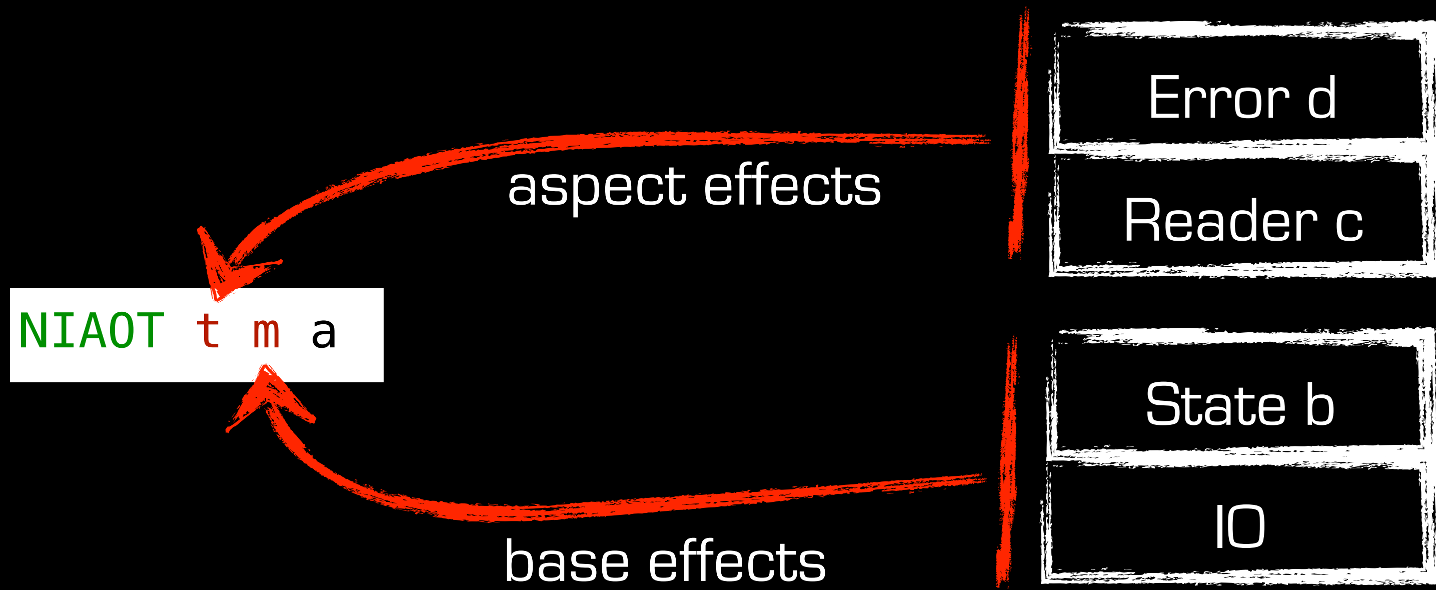
Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]



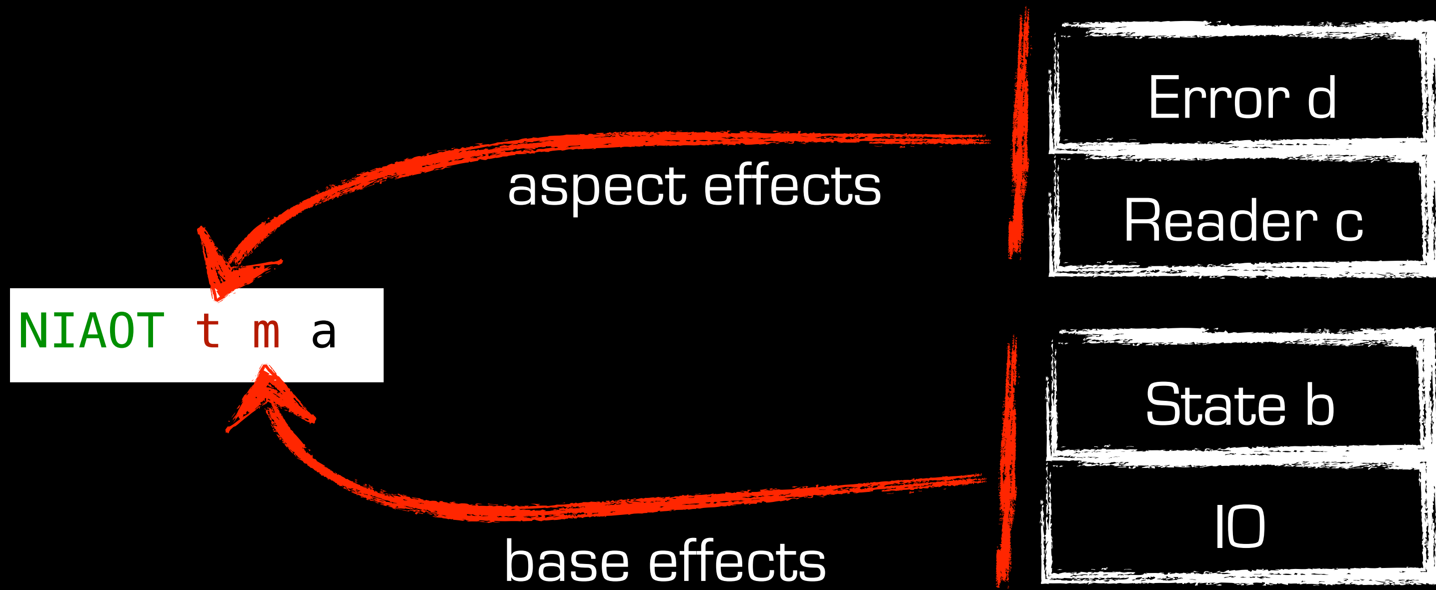
Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]



Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]



rely on parametricity to enforce non-interference

Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]

`NIAOT t m a`

aspect effects

Error d

Reader c

base effects

State b

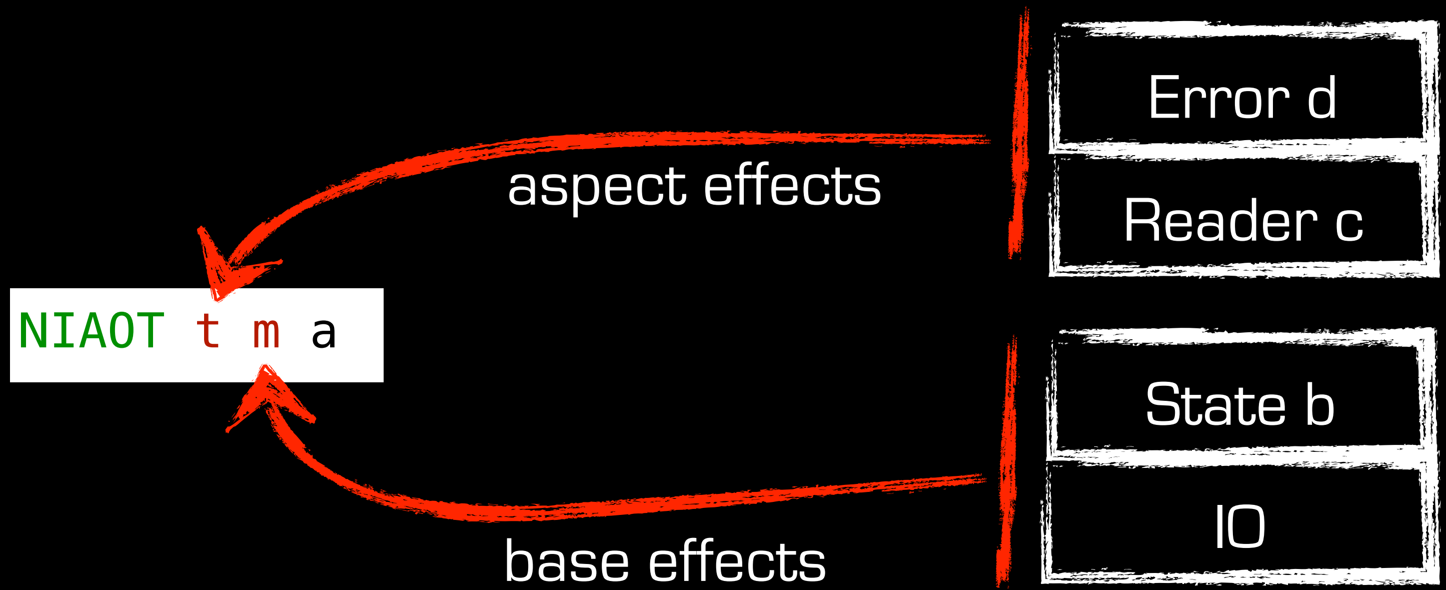
IO

rely on parametricity to enforce non-interference

```
type NIAdvice t a b = forall m. Advice (NIAOT t m) a b
```

Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]

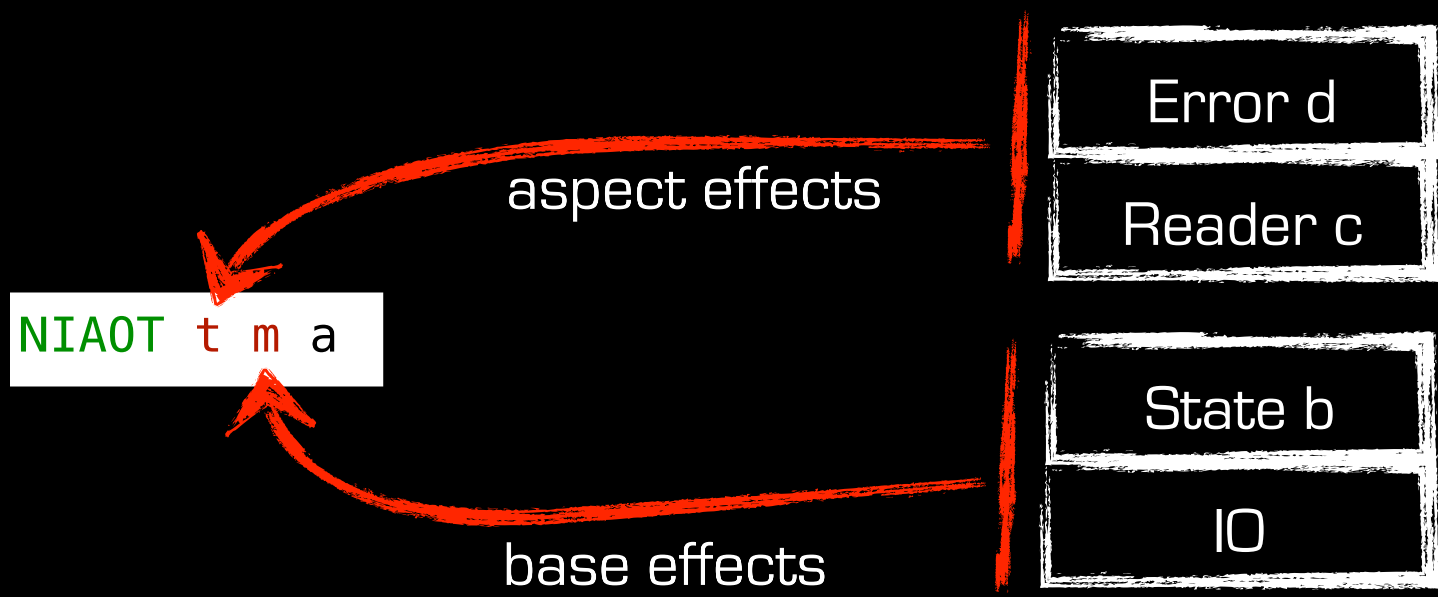


rely on parametricity to enforce non-interference

```
type NIAdvice t a b = forall m. Advice (NIAOT t m) a b
```

Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]



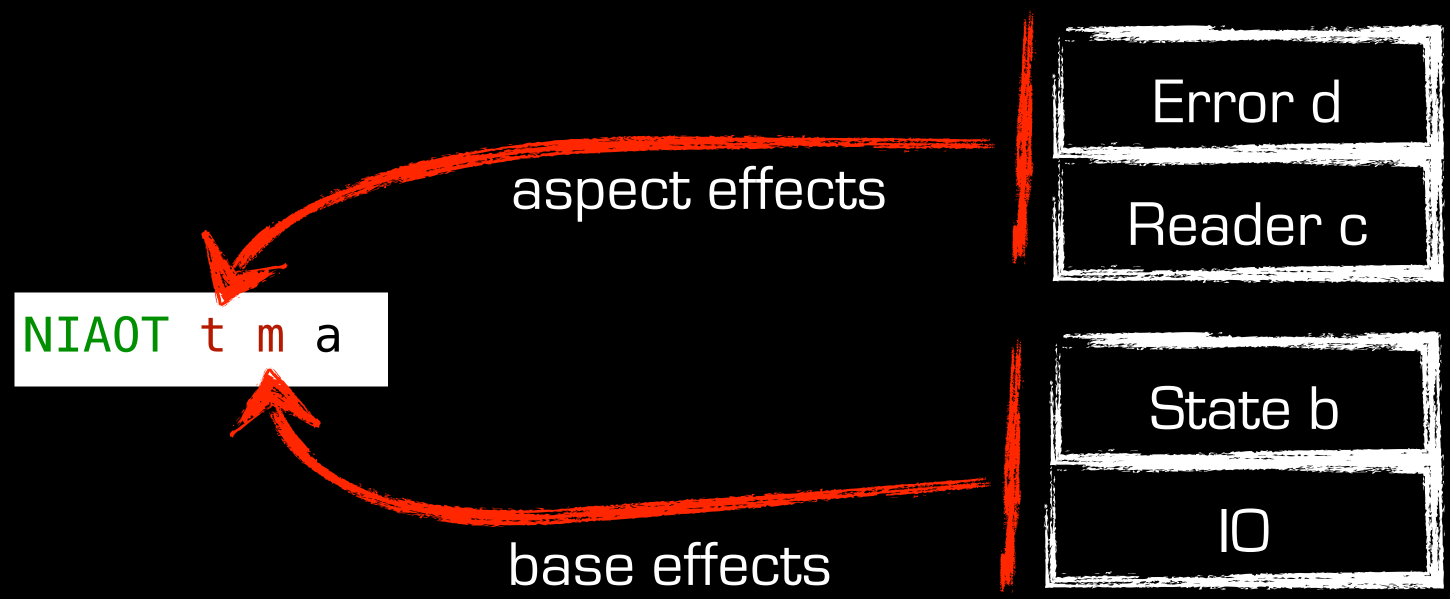
rely on parametricity to enforce non-interference

```
type NIAdvice t a b = forall m. Advice (NIAOT t m) a b

type NIPC t a b = forall m. PC (NIAOT t m) a b
```

Effect Interference

Reason about interferences base/aspects [Oliveira, 2010]



rely on parametricity to enforce non-interference

```
type NIAdvice t a b = forall m. Advice (NIAOT t m) a b

type NIPC t a b = forall m. PC (NIAOT t m) a b

type NIBase m a b = forall t. a -> NIAOT t m b
```

Enforcing Non-Interfering Advice

```
type NIAdvice t a b = forall m. Advice (NIAOT t m) a b
```

Enforcing Non-Interfering Advice

```
type NIAdvice t a b = forall m. Advice (NIAOT t m) a b
```

combinator that requires NIAdvice

```
niAdvice :: NIAdvice t a b -> Advice (NIAOT t m) a b
```

Enforcing Non-Interfering Advice

```
type NIAdvice t a b = forall m. Advice (NIAOT t m) a b
```

combinator that requires NIAdvice

```
niAdvice :: NIAdvice t a b -> Advice (NIAOT t m) a b
```

```
module Fib (fib, ppcFib) where
```

```
ppcFib = protectPC pcFib niAdvice
```

Enforcing Non-Interfering Advice

```haskell
type NIAdvice t a b = forall m. Advice (NIAOT t m) a b
```

combinator that requires NIAdvice

```haskell
niAdvice :: NIAdvice t a b -> Advice (NIAOT t m) a b
```

```haskell
module Fib (fib, ppcFib) where
```

```haskell
ppcFib = protectPC pcFib niAdvice
```

Enforcing Non-Interfering Advice

```
type NIAdvice t a b = forall m. Advice (NIAOT t m) a b
```

combinator that requires NIAdvice

```
niAdvice :: NIAdvice t a b -> Advice (NIAOT t m) a b
```

```
module Fib (fib, ppcFib) where
```

```
ppcFib = protectPC pcFib niAdvice
```

```
memoize :: NIAdvice ...
memoize = ...
```

Perspectives

Perspectives

extend EffectiveAdvice to deal with **quantification**

Perspectives

extend EffectiveAdvice to deal with **quantification**

extend Open Modules to deal with **effects**

Perspectives

extend EffectiveAdvice to deal with **quantification**

extend Open Modules to deal with **effects**

Challenges

Perspectives

extend EffectiveAdvice to deal with **quantification**

extend Open Modules to deal with **effects**

Challenges

- beyond the base/aspects distinction

Perspectives

extend EffectiveAdvice to deal with **quantification**

extend Open Modules to deal with **effects**

Challenges

- beyond the base/aspects distinction
- compose restrictions (eg. non-interfering + narrowing)

Perspectives

extend EffectiveAdvice to deal with **quantification**

extend Open Modules to deal with **effects**

Challenges

- beyond the base/aspects distinction
- compose restrictions (eg. non-interfering + narrowing)
- type system challenges
  - higher-rank polymorphism
  - managing the monadic stack: views [Schrijvers, 2011]

Conclusions

Scoping

- balance flexibility / guarantees
- practical & efficient implementations
- new models

Scoping

- balance flexibility / guarantees
- practical & efficient implementations
- new models

Interfaces

- time to try them out for real
- need a gradual adoption path

Scoping

- balance flexibility / guarantees
- practical & efficient implementations
- new models

Interfaces

- time to try them out for real
- need a gradual adoption path

Typing

- Holy Grail: expressiveness vs. complexity

Scoping

- balance flexibility / guarantees
- practical & efficient implementations
- new models

Interfaces

- time to try them out for real
- need a gradual adoption path

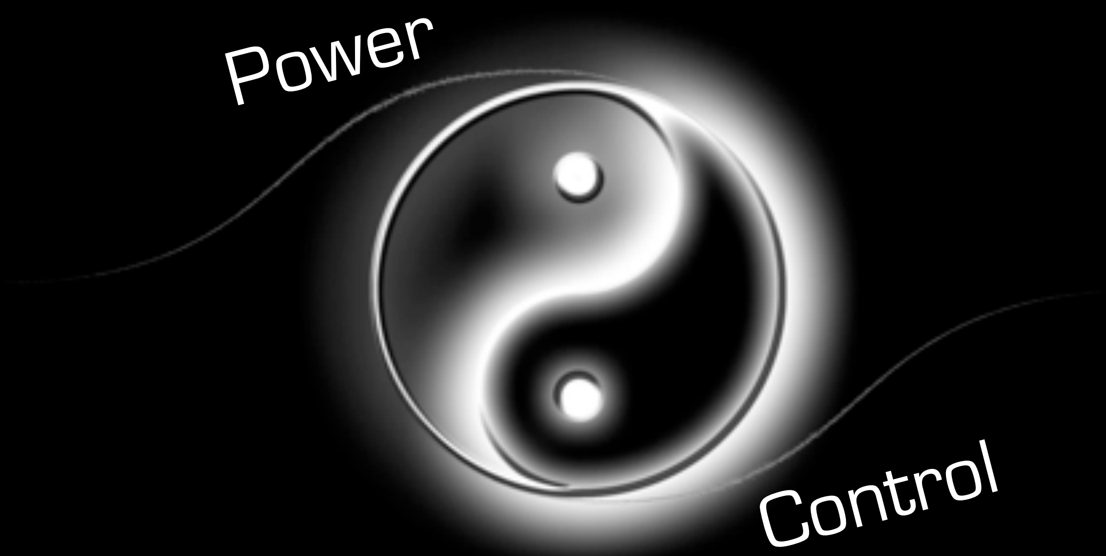
Typing

- Holy Grail: expressiveness vs. complexity

Effects

- exploit the (existing) type system or design specific analyses?
- lightweight & practical

TAMING
ASPECT
ORIENTATION



To be continued...

[Aldrich, 2005] Jonathan Aldrich: Open Modules: Modular Reasoning About Advice. ECOOP 2005:144-168

[Bagherzadeh, 2011] Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, Sean L. Mooney: Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. AOSD 2011:141-152

[Bodden, 2011] Eric Bodden: Closure joinpoints: block joinpoints without surprises. AOSD 2011:117-128

[Bodden, 2013] Eric Bodden, Éric Tanter, Milton Inostroza. Join Point Interfaces for Safe and Flexible Decoupling of Aspects. TOSEM, to appear.

[Boudol, 2004] Gérard Boudol: A Generic Membrane Model (Note). Global Computing 2004: 208-222

[Brichau, 2008] Johan Brichau, Andy Kellens, Kris Gybels, Kim Mens, Robert Hirschfeld, Theo D'Hondt: Application-specific models and pointcuts using a logic metalanguage. Computer Languages, Systems & Structures (CL) 34(2-3):66-82 (2008)

[De Fraine, 2008] Bruno De Fraine, Mario Südholt, Viviane Jonckers: StrongAspectJ: flexible and safe pointcut/advice bindings. AOSD 2008:60-71

[De Fraine, 2010] Bruno De Fraine, Erik Ernst, Mario Südholt: Essential AOP: The A Calculus. ECOOP 2010:101-125

[Figueroa, 2013] Ismael Figueroa, Nicolas Tabareau, Éric Tanter. Taming Aspects with Monads and Membranes. FOAL 2013:1-6

[Gudmundson, 2001] Stephan Gudmundson, Gregor Kiczales: Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. Workshop on Advanced Separation of Concerns 2001

[Gybels, 2003] Kris Gybels, Johan Brichau: Arranging language features for more robust pattern-based crosscuts. AOSD 2003:60-69

[Hoffman, 2012] Kevin Hoffman, Patrick Eugster: Trading Obliviousness for Modularity with Cooperative Aspect-oriented Programming. TOSEM, to appear.

[Inostroza, 2011] Milton Inostroza, Éric Tanter, Eric Bodden: Join point interfaces for modular reasoning in aspect-oriented programs. SIGSOFT FSE 2011: 508-511

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

[Katz, 2003] Shmuel Katz, Marcelo Sihman: Aspect Validation Using Model Checking. Verification: Theory and Practice 2003:373-394

[Kiczales, 1997] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, Gail C. Murphy: Open Implementation Design Guidelines. ICSE 1997:481-490

[Kiczales, 2005] Gregor Kiczales, Mira Mezini: Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. ECOOP 2005:195-213

[Krishnamurthi, 2004] Shriram Krishnamurthi, Kathi Fisler, Michael Greenberg: Verifying aspect advice modularly. SIGSOFT FSE 2004:137-146

[Moret, 2011] Philippe Moret, Walter Binder, Éric Tanter: Polymorphic bytecode instrumentation. AOSD 2011: 129-140

[Rajan, 2008] Hridesh Rajan, Gary T. Leavens: Ptolemy: A Language with Quantified, Typed Events. ECOOP 2008:155-179

[Rinard, 2004] Martin C. Rinard, Alexandru Salcianu, Suhabe Bugrara: A classification system and analysis for aspect-oriented programs. SIGSOFT FSE 2004: 147-158

[Oliveira, 2010] Bruno C. d. S. Oliveira, Tom Schrijvers, William R. Cook: EffectiveAdvice: disciplined advice with explicit effects. AOSD 2010: 109-120

[Ostermann, 2005] Klaus Ostermann, Mira Mezini, Christoph Bockisch: Expressive Pointcuts for Increased Modularity. ECOOP 2005: 214-240

[Schmitt, 2004] Alan Schmitt, Jean-Bernard Stefani: The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. Global Computing 2004: 146-178

[Schrijvers, 2011] Tom Schrijvers, Bruno C. d. S. Oliveira: Monads, zippers and views: virtualizing the monad stack. ICFP 2011: 32-44

[Skotiniotis, 2004] Therapon Skotiniotis, David H. Lorenz: Cona: aspects for contracts and contracts for aspects. OOPSLA Companion 2004: 196-197

[Steimann, 2010] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, Christian Kästner: Types and modularity for implicit invocation with implicit announcement. ACM Trans. Softw. Eng. Methodol. 20(1): (2010)

[Tabareau, 2013] Nicolas Tabareau, Ismael Figueroa, Éric Tanter. A Typed Monadic Embedding of Aspects. AOSD 2013:171-184.

[Tanter, 2008] Éric Tanter: Expressive scoping of dynamically-deployed aspects. AOSD 2008:168-179

[Tanter, 2009] Éric Tanter: Beyond static and dynamic scope. DLS 2009:3-14

[Tanter, 2010a] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, Mario Südholt: Scoping strategies for distributed aspects. Sci. Comput. Program. (SCP) 75(12):1235-1261 (2010)

[Tanter, 2010b] Éric Tanter: Execution levels for aspect-oriented programming. AOSD 2010:37-48

[Tanter, 2010c] Éric Tanter, Philippe Moret, Walter Binder, Danilo Ansaloni: Composition of dynamic analysis aspects. GPCE 2010: 113-122

[Tanter, 2012] Éric Tanter, Nicolas Tabareau, Rémi Douence: Taming aspects with membranes. FOAL 2012:3-8

[Toledo, 2011] Rodolfo Toledo, Éric Tanter: Access Control in JavaScript. IEEE Software 28(5): 76-84 (2011)

[Toledo, 2012] Rodolfo Toledo, Angel Núñez, Éric Tanter, Jacques Noyé: Aspectizing Java Access Control. IEEE Trans. Software Eng. 38(1): 101-117 (2012)

[Toledo, 2013] Rodolfo Toledo, Éric Tanter. Secure and Modular Access Control with Aspects. AOSD 2013:157-170

[Zhao, 2003] Jianjun Zhao, Martin C. Rinard: Pipa: A Behavioral Interface Specification Language for AspectJ. FASE 2003:150-165