10. Formal Models for Aspects

– Interpretation, Weaving and Compilation –

Topics Overview

- Why Formal Models?
  - Problems of AspectJ & Co.
- Semantics of Aspects
  - Interpreter semantics versus weaving semantics
  - Weaving semantics versus compilation semantics
  - Compiling aspects to Conditional Transformations
- Next Lecture: Use of Formal Models for Aspect Analysis
  - Interaction and Interference
  - Aspect-Aware-Refactorings

Why Formal Models?

Why Formal Models?

- Many different approaches
  - Pointcuts, Advice, Introductions → AspectJ
  - Plain Java APIs → AspectWerkz, JBoss/AOP, Prose
  - Aspect-Oriented Middleware → AspectWerkz, JBoss/AOP
  - Symmetric models → HyperJ, Composition Filters
  - Non-intrusive models → HyperJ
  - Encapsulation-respecting models → Composition Filters, Compose*
  - Dynamic aspects → ClassBoxes, Prose, Steamloom, …

Questions

- What is AOP?
  - Pointcuts
  - Joinpoints
  - Advices
  - Introductions
- But hat do these buzzwords really mean?
  - For instance, what is a Join Point?

Was ist eigentlich… - ein Joinpoint?

- Aspect-J-Definition des Joinpoint:

  „Ein Ereignis während der Programmausführung, … !„

  „Alle Prädikate eines Pointcut wirken mit bei der Selektion eines Joinpoints“

- Folgefrage 1
  - Was ist aber dann mit Introductions? LogicAJ hat doch offensichtlich JoinPoints für Introductions und die können kaum Ereignisse im Programmablauf sein…
- Folgefrage 2
  - Wenn es nur einen Joinpoint gibt, wie sage ich „Aufrufe die aus typen stammen, die in dieser Klasse instanziiert werden“?

Aspect-J:
```
call(* *.*(..)) && ... && call(*.new(..))
```

LogicAJ:
```
call(* ?CalledType.*(..)) && ... && call(?CalledType.new(..))
```

  - Beides funktioniert nicht… (Siehe nächstes Beispiel).

Beispiel: Design-Prinzip „Law of Demeter“

- Prinzip: „Talk only to your friends!“
  - Abhängigkeiten reduzieren, indem man in einer Klasse C nur Methoden aus "bekannten" Typen aufruft
- Als in Klasse C "bekannter" Typ gelten
  - die Typen der Instanzvariablen
  - die Typen der Methoden (Ergebnistypen)
  - die Typen der Methodenparameter
  - die Typen die im Code von C instantiiert werden
  - C selbst

Pointcut in_C_AufgerufeneMethode:

call(* *.*(..)) && withincode(* C.*(..))

Pointcut in_C_InstantiierteKlasse:

call(*.new(..)) && withincode(* C.*(..))

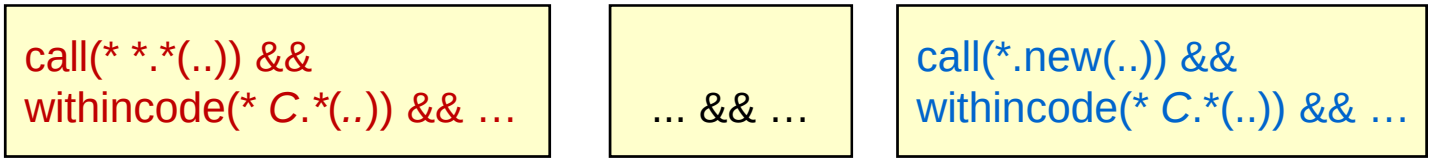
Aufgerufene Methode stammt aus bekanntem Typ →

Methode wird aufgerufen und stammt aus bekanntem Typ →

Methode wird aufgerufen und stammt aus Typ der (… in C instantiiert wird) →

call(* *.*(..)) && ... && call(*.new(..)) hat Schnittmenge call(*.new(..)) ☹

JoinPoints: Manchmal braucht man mehr!



- Problem
  - Verschiedene Joinpoints die sich nicht gegenseitig beeinflussen sind manchmal erforderlich
- Idee: Multi-Pointcut [LogicAJ]
  - Scope = Teil eines Pointcut der einen Join Point selektiert
  - Multi-Pointcut enthält mehrere Scopes, die verschiedene Join Points ansprechen
- Zusammenspiel des Pointcuts mit Body von Advice bzw. Introduction
  - a) Advice Body bezieht sich auf einen ausgesuchten Joinpoint oder
  - b) Verschiedene Scopes im Advice beziehen sich auf die verschiedenen vom Multi-Pointcut selektierten Join Points

Law of Demeter in LogicAJ und LogicAJ2

```
1.  aspect LawOfDemeter {
2.
3.   pointcut knownTo(?CallingType, ??ParamTypes, ?Type) :
4.     equals(?Type, ?CallingType)                        // rule 1
5.     || member(?Type, ??ParamTypes)                     // rule 2
6.     || field(?Type ?CallingType.?Fname)                // rule 3
7.     || method(?Type ?CallingType.?Mname(..))           // rule 4
8.     || ( call(?ConstrCall, ?Type.new(..)) &&           // rule 5
9.          withincode(?ConstrCall, * ?CallingType.*(..))
10.       );
11.
12. declare warning :
13.  call(* ?CalledType.?CalledMeth(..)) &&
14.  withincode(* ?CallingType.?CallingMeth(??ParamTypes)) &&
15.  !(knownTo(?CallingType,??ParamTypes,?Type) &&     // rule 1-5
16.  subtype(?Type, ?CalledType))                       // rule 6
17.  :
18.  "The call violates the Law of Demeter.";
19.}
```

- Die Pointcuts in Zeile 8 und 9 selektieren einen anderen Join Point als der Rest
- Dies wird explizit gemachten durch den zusätzlichen ersten Parameter ?ConstrCall der beiden Pointcuts
- Der Advice bezieht sich weiterhin auf den impliziten Join Point der in Zeile 4-7 und 13-14 selektiert wird.

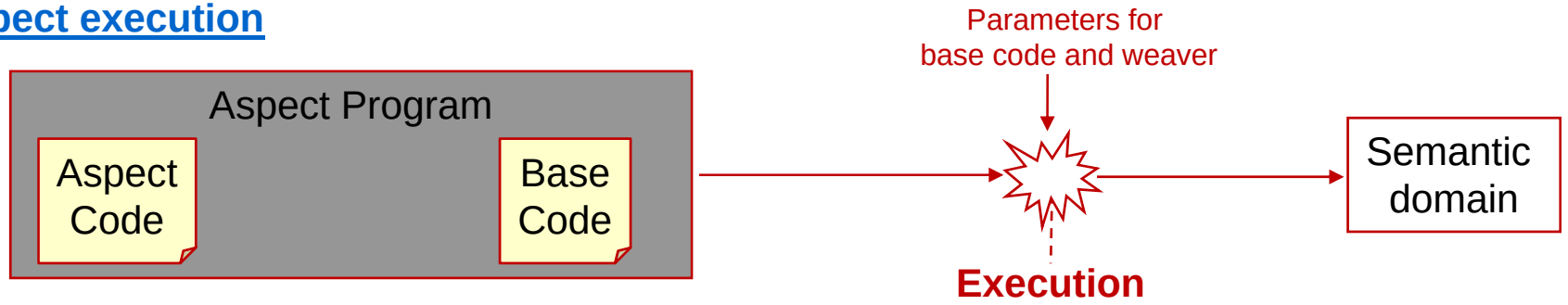
Fazit

- Kernbegriffe teilweise unklar, z.B. Begriff des Join-Points
  - Join Points für Introductions sind keine Ereignisse
  - Singuläre Join-Points versus Multi-Join-Points
- Was kann uns helfen, herauszufinden worum es hier geht?
- ➢ Formale Modelle

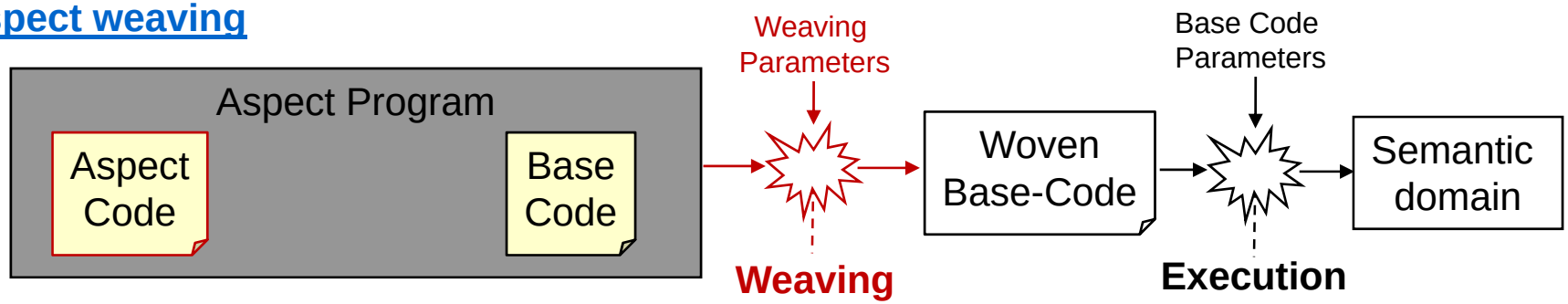
Interpreter Semantics via „Natural Semantics“

Approaches to Aspect Semantics

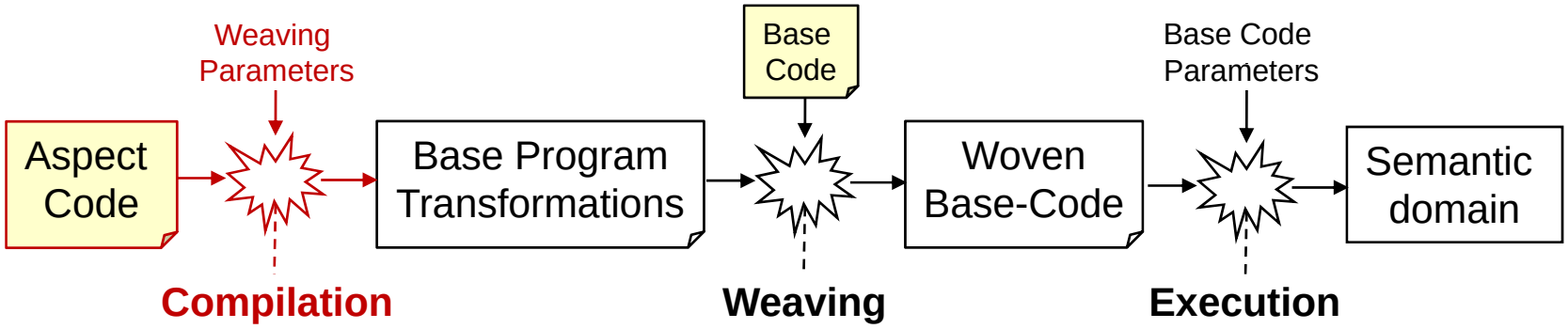
Aspect execution



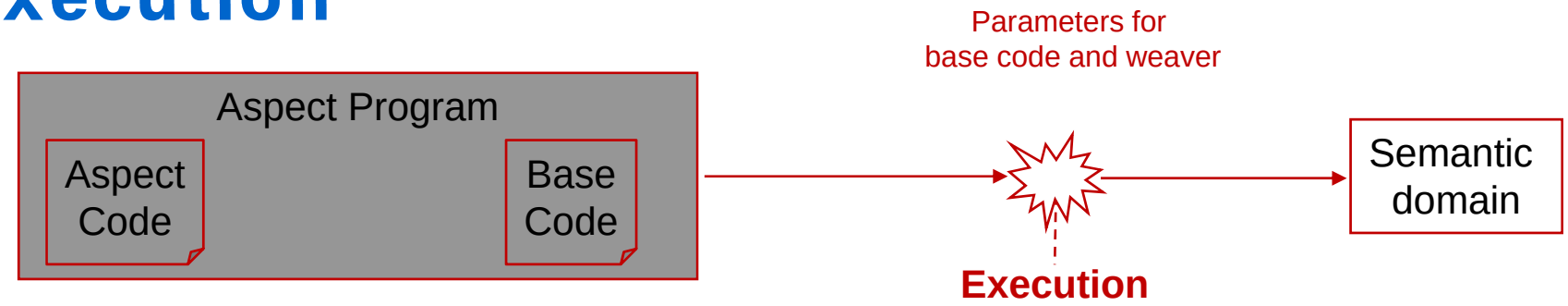
Aspect weaving



Aspect compilation



Approaches to Aspect Semantics: "Aspect Execution"



- Need to define
  - Language Syntax
  - Well-formed expressions
  - Typing (static semantics)
    - ⇨ Types
    - ⇨ Type environments
    - ⇨ Typing rules rules
  - Execution (dynamic semantics)
    - ⇨ Run-time values
    - ⇨ Store
    - ⇨ Derivation (Interpretation)

Typically done using "Natural Semantics"

Natural Semantics

- Natural Semantics is based on Plotkin's Structural Operational Semantics (SOS) and further developed at INRIA by Kahn.
- Specifications consist of
  - data type declarations (abstract syntax, environments, run-time values, types, etc.) and
  - sets of inference rules.
- The inference rules specify relations between objects, in a style akin to Gentzen‘s Sequent Calculus for Natural Deduction. (Hence the name Natural Semantics.)

- This is just to give you an starting point for own further investigation of the basics.
- Focus here
  - Basic intro
  - Use of natural semantics for defining method call interception

Roots of „Natural Semantics“

- http://www-sop.inria.fr/tropics/Laurent.Hascoet/papers/CDDHK85.html

- @inproceedings{ CDDHK85,
  - author = {Clement, D. and Despeyroux, J. and Despeyroux, T. and Hascoet, L. and Kahn, G.},
  - title = {Natural Semantics on the computer},
  - booktitle = {K. Fuchi and M. Nivat, editors, proceedings of the France-Japan AI and CS Symposium, ICOT, Japan},
  - pages = {49-89},
  - note = {also Technical Memorandum PL-86-6 Information Processing Society of Japan and Rapport de recherche \#0416, INRIA},
  - url = "http://www.inria.fr/rrrt/rr-0416.html", year = 1986

  }

Natural Semantics: Values

- Definition of basic values
  - Integers, booleans, ...
- Definition of language syntax
  - Example:

$$
\begin{array}{rcl}
AExp & ::= & \overline{n} \mid x \mid (a_0 \oplus a_1) \\
BExp & ::= & \mathbf{true} \mid \mathbf{false} \mid (a_0 \odot a_1) \mid (b_0 \oslash b_1) \mid (\neg b) \\
Com & ::= & \mathbf{skip} \mid x := a \mid (c_0 \; ; \; c_1) \mid (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \mid (\mathbf{while}\ b\ \mathbf{do}\ c\ ) \\
\oplus & ::= & + \mid * \mid - \\
\odot & ::= & \leq \mid = \\
\oslash & ::= & \vee \mid \wedge
\end{array}
$$

- Stores
  - A function that assigns a value to each variable.
- Configurations
  - A snapshot of an executing program represented as a pair <expr,store>
  - 'store' represents the current values of the variables
  - 'expr' represents the next command to be executed

Natural Semantics: Small versus Big Step Semantics

Small Step

- Specifies the operation of a program one step at a time.
- Small-step reduction
  - $<e_1, s_1> \rightarrow < e_2, s_2>$
  - $<e_1, s_1>$ reduces to $< e_2, s_2>$ in one step
- Example: Semantics of sequence

$$\frac{\langle c_0, \sigma\rangle \rightarrow \langle c_0', \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \rightarrow \langle c_0'; c_1, \sigma'\rangle} \qquad \frac{}{\langle \mathbf{skip}; c_1, \sigma\rangle \rightarrow \langle c_1, \sigma\rangle}$$

- Continue to apply rules to configurations until reaching a final configuration <skip, store>.
- Indirect reduction
  - $<e_1, s_1> \rightarrow^* < e_2, s_2>$
  - $<e_1, s_1>$ reduces to $< e_2, s_2>$ in zero or more steps

Big Step

- Specifies the entire transition from a configuration to a final value.
- Big-step reduction
  - $<e_1, s_1> \Downarrow s_{final}$
  - $<e_1, s_1>$ reduces to s_{final} in possibly many steps
- Example: Semantics of sequence

$$\frac{\langle c_0, \sigma\rangle \Downarrow \sigma' \quad \langle c_1, \sigma'\rangle \Downarrow \sigma''}{\langle c_0; c_1, \sigma\rangle \Downarrow \sigma''}$$

- Finished after applying one rule
  - Contains applications of all the others

Big-Step Semantics for „Method Call Interception“

Ralf Lämmel

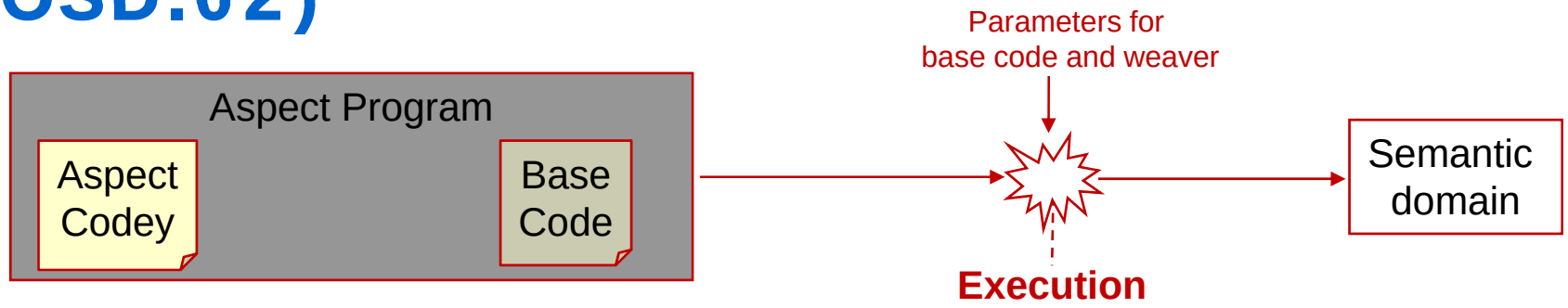
Proceedings of first International Conference on

Aspect-Oriented Software Development

(AOSD 2002)

universitätbonn ROOTS

Method Call Intrception (Ralf Lämmel, AOSD.02)



- An object-oriented core language
  - Syntax
  - Static semantics
  - Dynamic Semantics
- Method call interception (MCI) ← An abstraction for advice
  - Syntax
  - Static semantics
  - Dynamic Semantics

OO Core Language: Static semantics (Typing)

Im aktuellem Kontext gilt "exp den Typ τ"

Well-typedness of expressions

$$T, \Sigma, \theta, \eta \vdash exp : \tau$$

Inferenzregel

Was als bekannt vorausgesetzt wird

$$\frac{T, \Sigma, \theta, \eta \vdash exp_1 : \tau_1 \quad \wedge \quad T, \Sigma, \theta, \eta \vdash exp_2 : \tau_2}{T, \Sigma, \theta, \eta \vdash exp_1; exp_2 : \tau_2} \quad [\text{seq}]$$

Was sich daraus ableiten lässt

- Bedeutung obiger Regel
  - "Der Typ einer Sequenz von Statements ist der Typ des letzten Statements in der Sequenz".

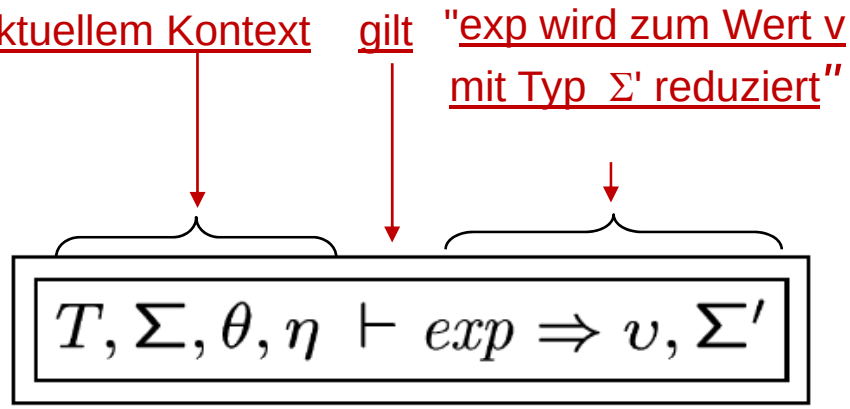
OO Core Language: Static semantics (Typing)

Static semantics of method calls

$$\frac{\begin{array}{lll}(1) & & T, \Sigma, \theta, \eta \vdash exp : cn \\ (2) & \wedge & \pi_1(T) @ \langle cn, mn \rangle = \langle \langle \tau_1, \ldots, \tau_n \rangle, \tau \rangle \\ (3) & \wedge & T, \Sigma, \theta, \eta \vdash exp_1 : \tau_1 \\ & \wedge & \ldots \\ & \wedge & T, \Sigma, \theta, \eta \vdash exp_n : \tau_n \end{array}}{T, \Sigma, \theta, \eta \vdash exp.mn\,(\,exp_1, \ldots, exp_n\,) : \tau} \quad [\text{call}]$$

- Bedeutung obiger Regel
  - "Der Typ eines Methodenaufrufs ist der Ergebnistyp der für die aufgerufene Methode in dem statischen Typ des Empfängers deklariert ist".

OO Core Language: Dynamic semantics (Execution)



Expression evaluation

$$T, \Sigma, \theta, \eta \vdash \mathbf{this} \Rightarrow \theta, \Sigma \qquad \text{[this]}$$

$$\frac{T, \Sigma_0, \theta, \eta \vdash exp_1 \Rightarrow v_1, \Sigma_1 \quad \wedge \quad T, \Sigma_1, \theta, \eta \vdash exp_2 \Rightarrow v_2, \Sigma_2}{T, \Sigma_0, \theta, \eta \vdash exp_1; exp_2 \Rightarrow v_2, \Sigma_2} \qquad \text{[seq]}$$

- Bedeutung obiger Regel
  - "Das Ergebnis einer Sequenz von Statements ist das Ergebnis des letzten Statements in der Sequenz".

OO Core Language: Dynamic semantics (Execution)

- Dynamic semantics of method calls

$$\frac{\begin{array}{lll}(1) & & T, \Sigma_0, \theta, \eta \vdash exp \Rightarrow \rho, \Sigma_1 \\ (2) & \wedge & \pi_1(T) @ \langle \rho, mn \rangle = \langle \langle vn_1, \ldots, vn_n \rangle, exp' \rangle \\ (3) & \wedge & T, \Sigma_1, \theta, \eta \vdash exp_1 \Rightarrow v_1, \Sigma_2 \\ & \wedge & \ldots \\ & \wedge & T, \Sigma_n, \theta, \eta \vdash exp_n \Rightarrow v_n, \Sigma_{n+1} \\ (4) & \wedge & \eta' = \bot[vn_1 \mapsto v_1, \ldots, vn_n \mapsto v_n] \\ (5) & \wedge & T, \Sigma_{n+1}, \rho, \eta' \vdash exp' \Rightarrow v, \Sigma_{n+2}\end{array}}{T, \Sigma_0, \theta, \eta \vdash exp.mn\ (\ exp_1, \ldots, exp_n\ ) \Rightarrow v, \Sigma_{n+2}}$$ [call]

- Bedeutung obiger Regel
  - "Das Ergebnis eines Methodenaufrufs ist das Ergebnis des Methodenbodies wenn man ihn in einem Kontext auswertet in dem die Argumente an die Werte gebunden sind zu denen die Parameter-Ausdrücke ausgewertet werden".

Method Call Interception (MCI)

Syntax of basic MCI

$$
\begin{array}{lcl}
exp & = & \cdots \mid \textbf{superimpose } exp \textbf{ on } eve \\
eve & = & mci\ loc \mid eve \textbf{ within } loc \\
loc & = & exp.mn \\
mci & = & \textbf{dispatch} \mid \textbf{enter} \mid \textbf{exit}
\end{array}
$$

The one and only example

$$\textbf{superimpose } counter.inc() \textbf{ on enter } b.mb \textbf{ within } a.ma$$

Method Call Interception (MCI)

- Static model of basic MCI
  - Rules for core language preserved
  - Additional rule for typing of **'superimpose'**
- Dynamic model of basic MCI
  - Extend judgement
  - Additional rule for execution of **'superimpose'**
  - Adaptation of execution of method calls

$$
\begin{array}{lcll}
\lambda & = & \rho \times mn & \text{(Locations)} \\
\Sigma & = & \cdots \times \boxed{\Delta} & \text{(Extended store)} \\
\Delta & = & \kappa \rightarrow_{fin} \alpha^* & \text{(MCI registry)} \\
\kappa & = & mci \times \lambda + \kappa \times \lambda & \text{(Events)} \\
\alpha & = & \lambda \times exp & \text{(Type-safe advice)} \\
\theta & = & \cdots \times \boxed{mn} \; (= \lambda) & \text{(Location awareness)}
\end{array}
$$

Basic MCI: Typing of Superimposition

Well-typedness of expressions

$$T, \Sigma, \theta, \eta \vdash exp : \tau$$

$$\frac{\begin{array}{rl} \wedge & T, \Sigma, \theta, \eta \vdash eve \\ \wedge & T, \Sigma, \theta, \bot \vdash exp : \tau \end{array}}{T, \Sigma, \theta, \eta \vdash \begin{array}{l} \textbf{superimpose } exp \\ \textbf{on } eve \end{array} : \textbf{void}} \quad \text{[superimpose]}$$

- This says: "Superimposition has type **void**"

Basic MCI: Typing of Events

Well-typedness of events

$$\boxed{T, \Sigma, \theta, \eta \vdash eve}$$

$$\frac{\begin{array}{rl} & T, \Sigma, \theta, \eta \vdash exp : cn \\ \wedge & \pi_1(T) @ \langle cn, mn \rangle = \_ \end{array}}{T, \Sigma, \theta, \eta \vdash mci\ exp.mn} \quad \text{[mci]}$$

$$\frac{\begin{array}{rl} & T, \Sigma, \theta, \eta \vdash eve \\ \wedge & T, \Sigma, \theta, \eta \vdash exp : cn \\ \wedge & \pi_1(T) @ \langle cn, mn \rangle = \_ \end{array}}{T, \Sigma, \theta, \eta \vdash eve\ \mathbf{within}\ exp.mn} \quad \text{[within]}$$

Basic MCI: Expression Evaluation

Expression evaluation

$$\boxed{T, \Sigma, \theta, \eta \vdash exp \Rightarrow v, \Sigma'}$$

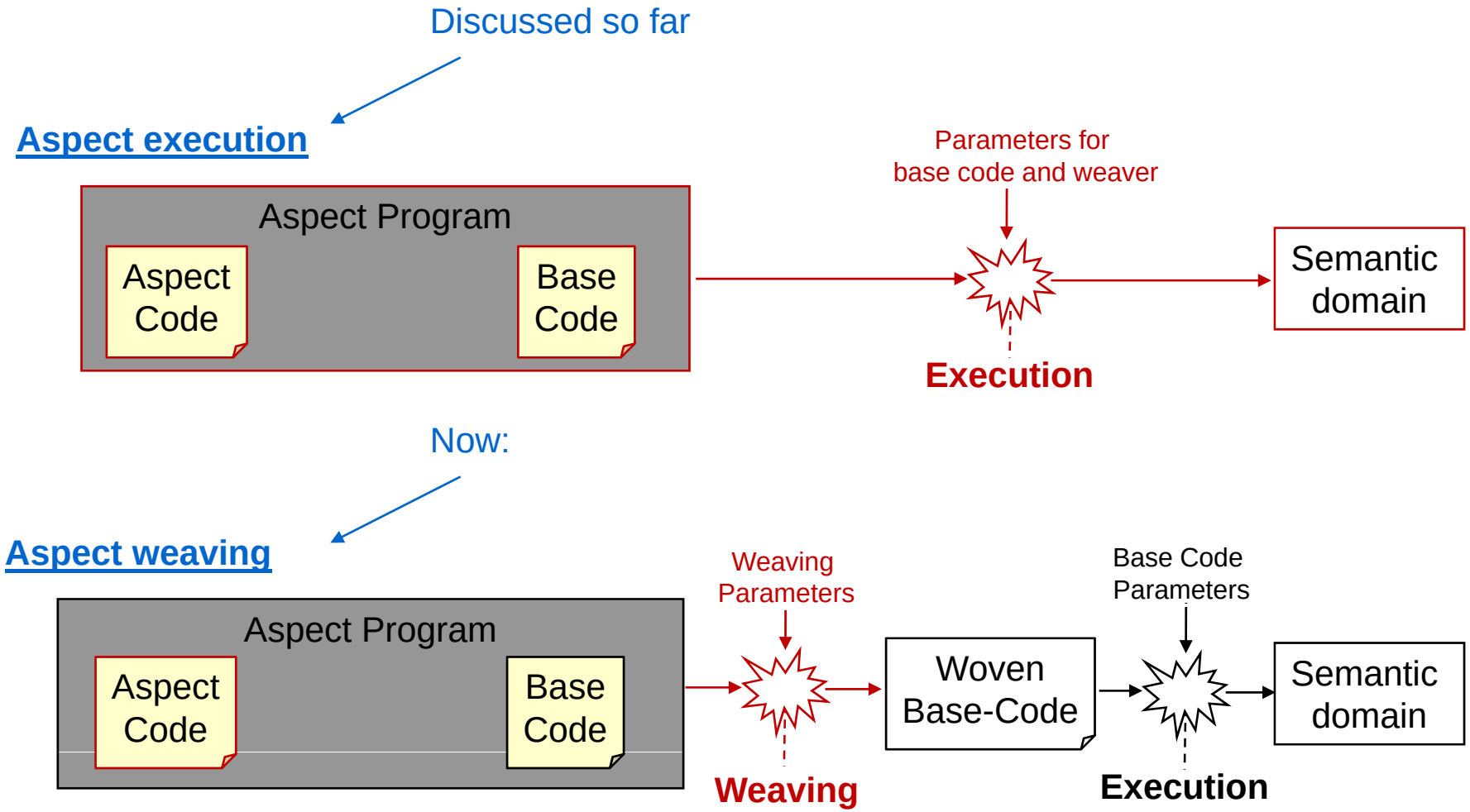
$$\frac{\begin{array}{ll} & T, \Sigma, \theta, \eta \vdash eve \Rightarrow \kappa, \Sigma' \\ \wedge & \alpha = \langle \theta, exp \rangle \\ \wedge & \text{register}(\Sigma', \kappa, \alpha) \Rightarrow \Sigma'' \end{array}}{T, \Sigma, \theta, \eta \vdash \begin{array}{r} \textbf{superimpose } exp \\ \textbf{on } eve \end{array} \Rightarrow 0, \Sigma''}$$

[superimpose]

Summary of MCI and friends (= formal models for aspect execution)

- Advantages
  - Focus on the core concepts → no distraction by language details
  - Immediate model for implementation of an interpreter
- Disadvantages
  - Might abstract too much → e.g. no semantics for introductions in MCI
  - Need to define all the relevant semantics of the core language
    - ⇨ ... and update it in all the places influenced by aspects
- Question
  - Could we define just the Aspect part?
  - ... Incrementally (without modifying the base language semantics)?
- Idea
  - Take base language semantics for granted.
  - Only specify semantics of weaving as translation of aspect program to pure base program.

Approaches to Aspect Semantics



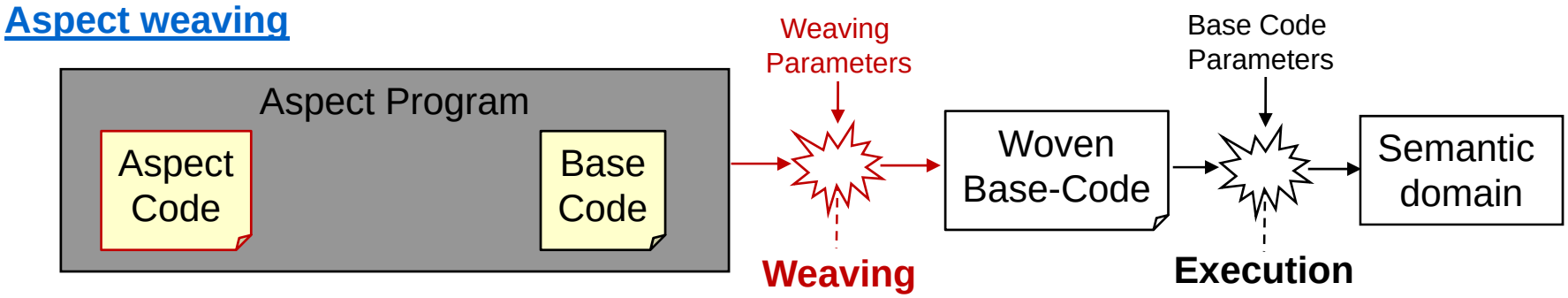
Aspect Weaving

Main idea

Weaving static versus dynamic join points

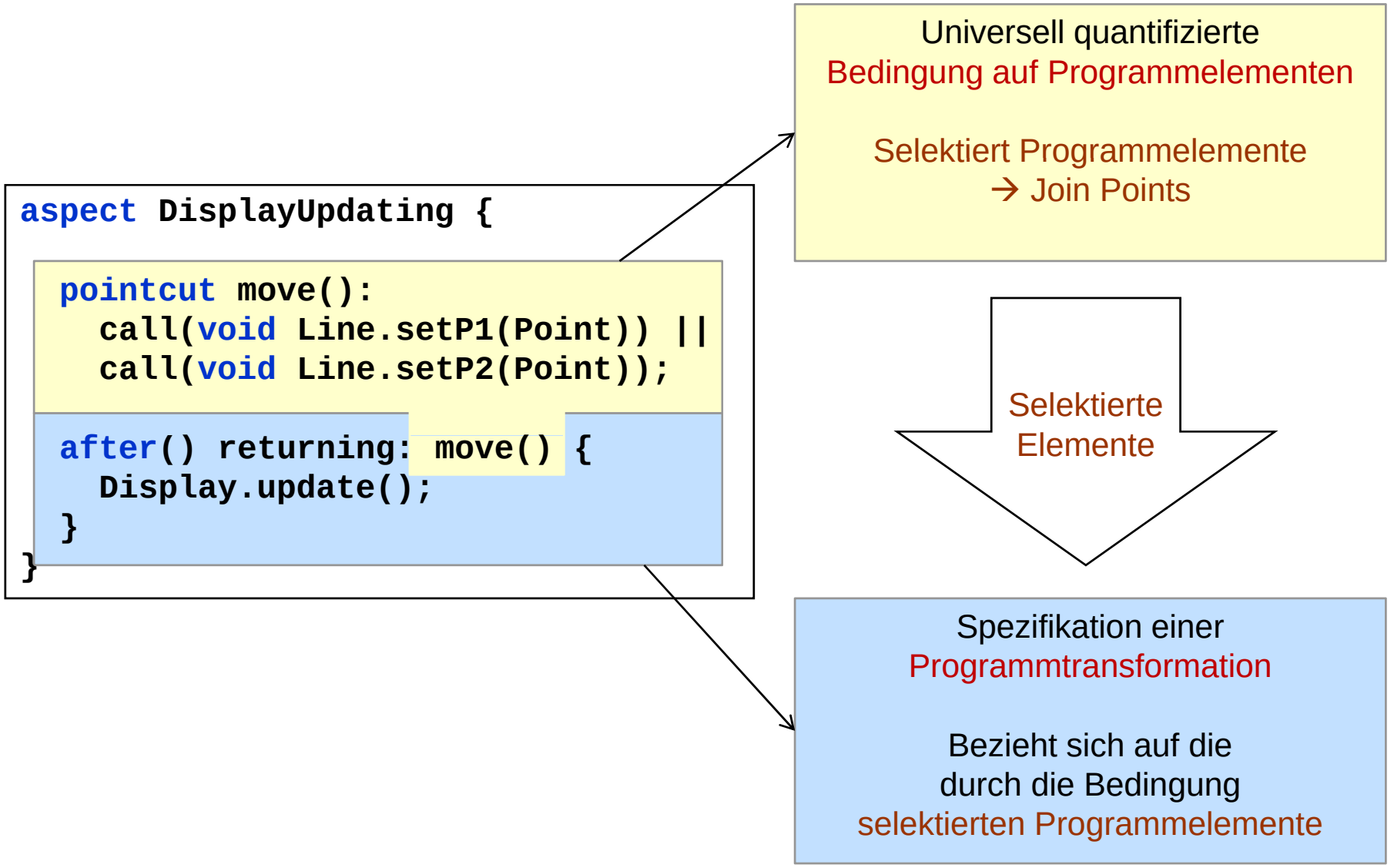
Weaving statically, at load-time or dynamically

Aspect Weaving Semantics



- Approach
  - Specify translation of aspect program to pure base program
  - Example: Translation of AspectJ to pure Java program
- Advantage
  - No need to modify / specify semantics of base language

Aspect Weaving



Dimensionen von Aspect-Weaving

- Was (wird übersetzt)
  - Statische pointcuts
    - ⇨ Z.B. "withincode"
  - Dynamische pointcuts
    - ⇨ Z.B. "cflow"

- Wann (wird übersetzt)
  - Vor dem Laden → static weaving
  - Während des Ladens → load-time weaving
  - Während der Ausführung → dynamic weaving

- Resultierende Problem-Gruppen

Aspect Weaving	Statische pointcuts	Dynamische pointcuts
Static weaving	?	?
Load-time weaving	?	?
Dynamic weaving	?	?

Statisches Weben statischer Pointcuts

- Einfach, weil
  - pointcut statisch ausgewertet werden kann
  - advice-code bereits während des weaving eingefügt werden kann

```
before : call(* *.get*(..)) &&
         withincode(* C.*(..)) {
   foo();
}
```

Weaving

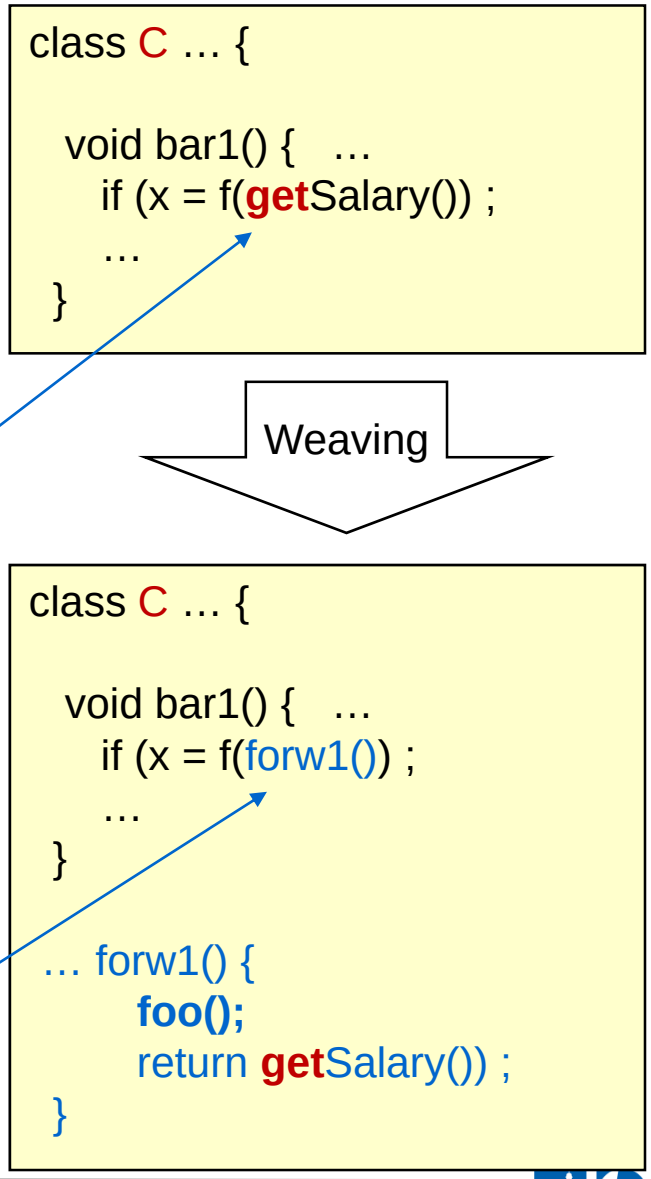
```
class C … {

  void bar1() {  …

     x = getSalary();
     …
  }

  void bar2() {  …

     y = getFood();
     …
  }
  …
}
```

Weaving

```
class C … {

  void bar1() {  …
     foo();  // ← vom Weaver eingefügt
     x = getSalary();
     …
  }

  void bar2() {  …
     foo();  // ← vom Weaver eingefügt
     y = getFood();
     …
  }
  …
}
```

Statisches Weben statischer Pointcuts: Komplikationen

- Geschachtelte Joinpoints
  - Variablenzugriffe und Methodenaufrufe
  - … innerhalb von Ausdrücken
- Problem
  - Es gibt evtl. keinen Punkt im Programmablauf, wo man den Aspektcode einweben könnte
  - → "Statements" können nicht mitten in "Expressions" stehen.
- Lösung
  - Forwarding-Methode einführen, die den Join Point und den Advice-Code enthält
  - Pointcut wird durch Aufruf der Forwarding-Method ersetzt



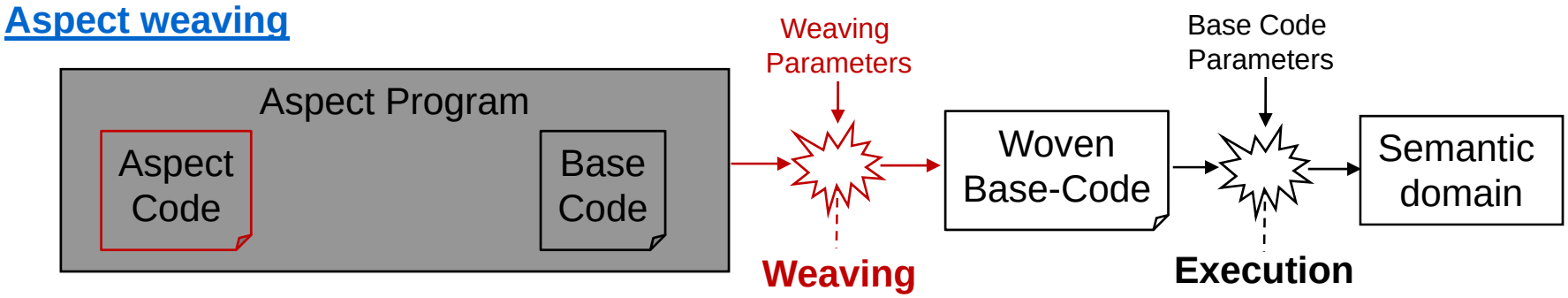
Statisches Weben Dynamischer Pointcuts

- args, target, …
  - statisch möglich
  - einfach einsetzen der entsprechenden Variablen
  - aus Sicht des Weaving also gar keine "dynamischen" pointcuts
- cflow – naiv
  - Simulation des Aufrufstacks pro thread
    - ⇨ z.B. durch Nutzung von "threadlocal"-Variablen in Java
  - Einfügen von Test-Code an jedem Join-Point-Shadow
    - ⇨ Anhand des Stacks überpfüfen, ob man sich tatsächlich im entsprechenden "cflow" befindet
  - Recht teuer!!!
- cflow – tuned
  - Je eine „threadlocal“ Variable pro cflow → Zähler für „unfertige“ Aufrufe
  - Zählen der Aufrufe (+1) und Returns (-1)
  - Man ist im cflow von Methode M wenn der Zähler für M > 0 ist.

Weaving in AspectJ

- Eigener Compiler
  - nutz Java API für Programmaanalyse (auf Byte-Code Ebene)
  - nutzt Java API für Manipulation von Byte-Code
  - mischt das Ganze mit Compilierung von Java-Code
- Frage
  - Bessere Trennung der Belange?

Aspect Weaving Semantics

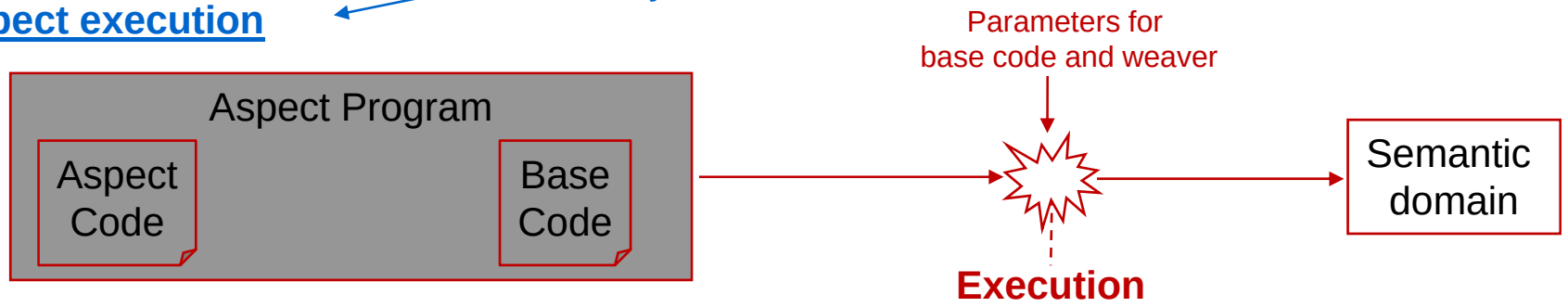


- Approach
  - Specify translation of aspect program to pure base program
- Advantage
  - No need to modify / specify semantics of base language
- Question
  - Can we achieve better separation of concerns?
  - Can we separately specify how the base program is modified and what it means to modify a program?

Approaches to Aspect Semantics

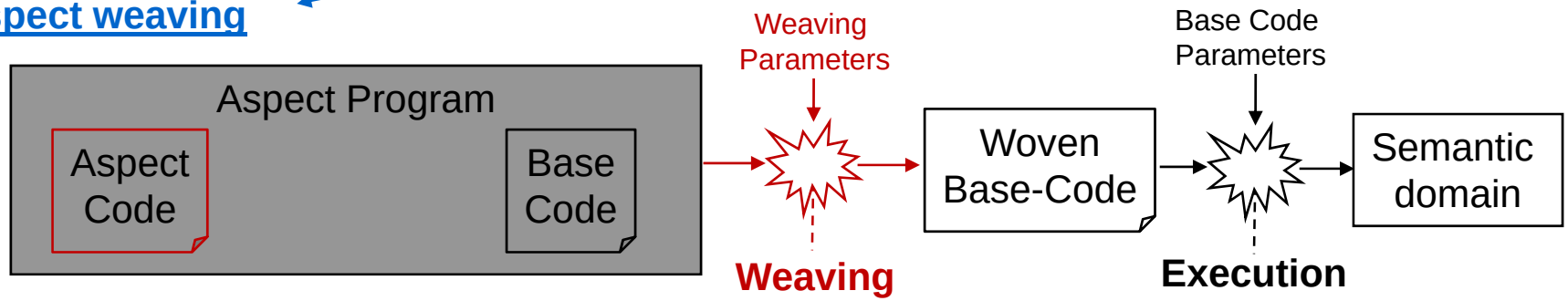
Already discussed

Aspect execution



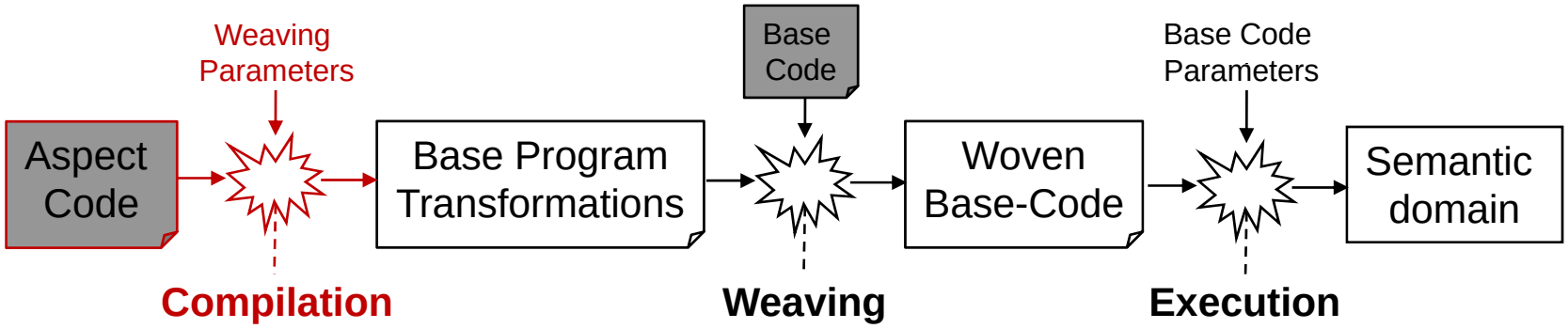
Already discussed

Aspect weaving

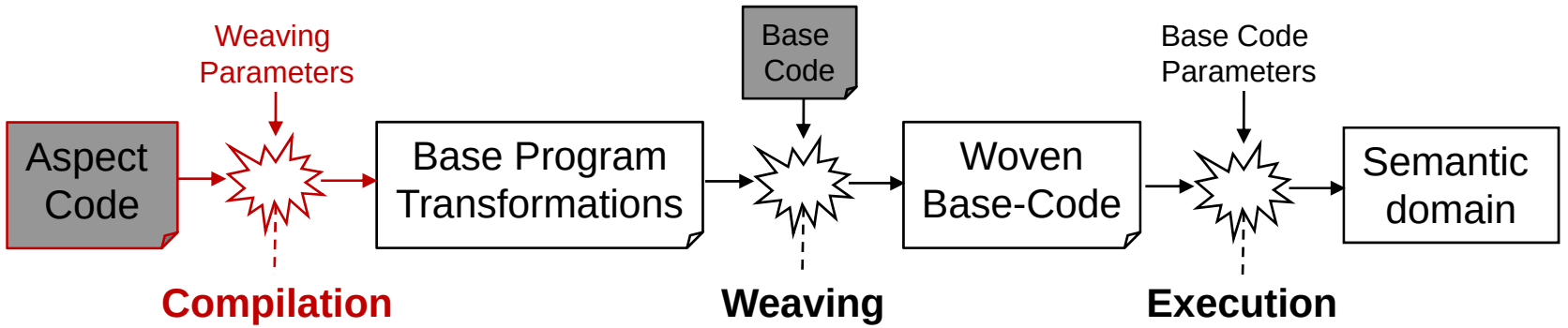


Next

Aspect compilation



Aspect Compilation



- Idea
  - Separate compilation from weaving
  - Aspect compilation translates aspects to specification of program transformations
- Advantage
  - Semantics of program transformation can be specified separately once and for all
  - Only compilation must be specified again for every pair of aspect language and base language

➔ Next
  - How to specify conditional program transformations
  - How to translate aspects to conditional program transformations

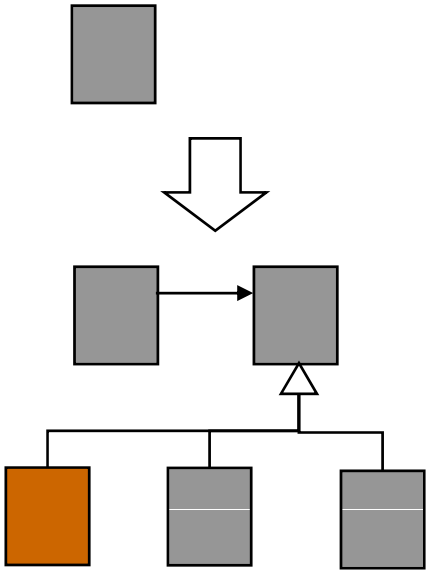
universitätbonn ROOTS

Conditional Program Transformations

Software Evolution

Refactoring

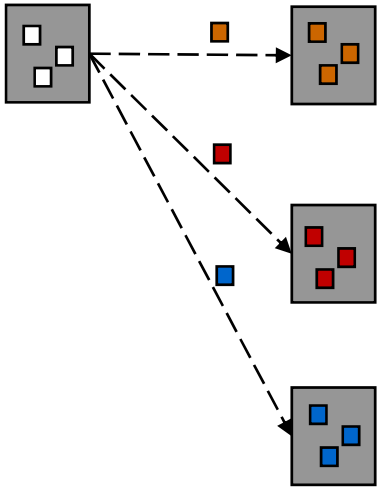
→ ease later evolution



→ **preconditions**

Generative Programming

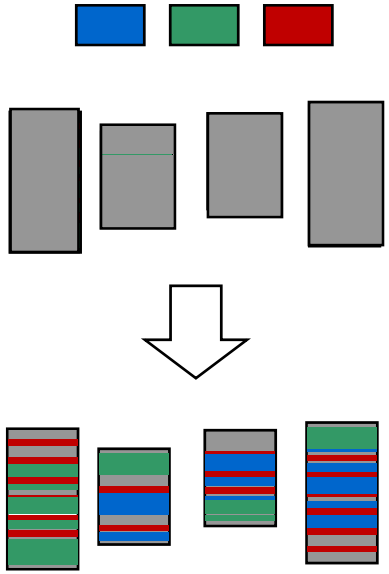
→ product lines



→ **parametrization**

Aspect-oriented programming

→ cross-cuting concerns

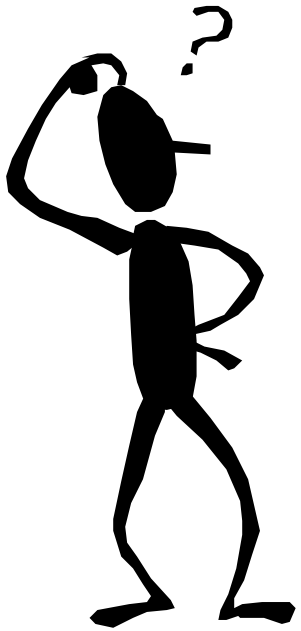


→ **joinpoints**

Software Evolution

Refactoring

→ ease later evolution



Generative Programming

→ product lines

Aspect-oriented programming

→ cross-cuting concerns

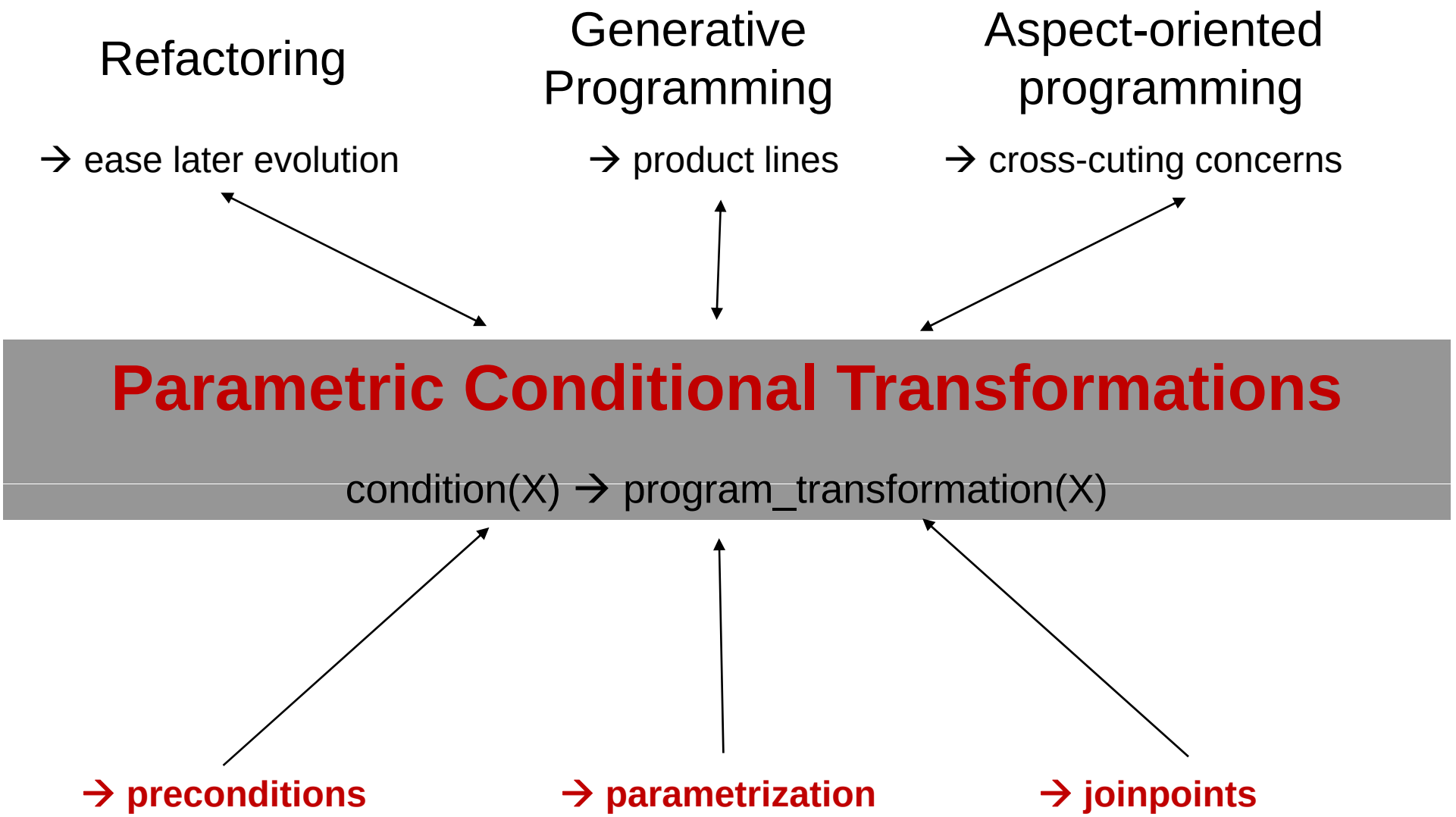
Unifying principle?

→ preconditions

→ parametrization

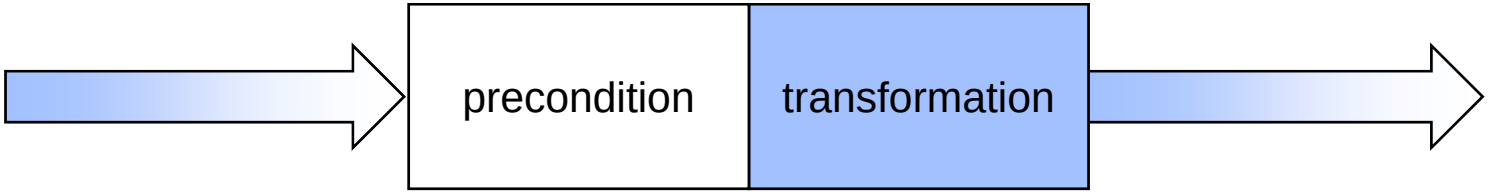
→ joinpoints

Software Evolution

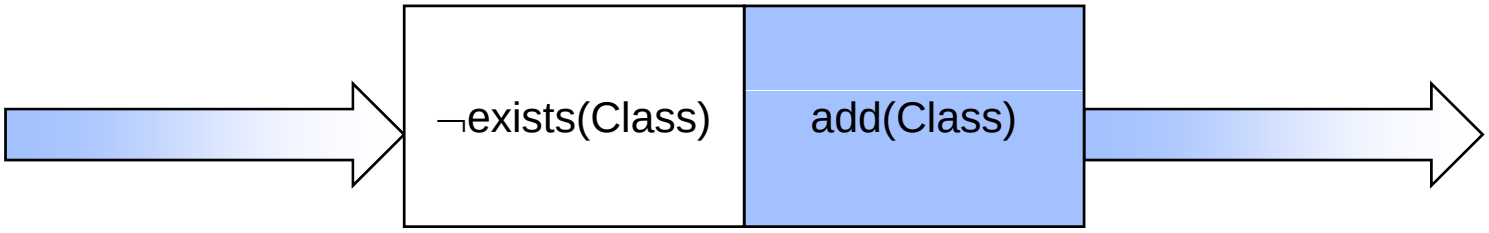


Conditional Transformations (CTs)

- CT = Precondition + Transformation
  - Precondition is true ⇒ Transformation may be executed



- Example: „Add Class" Transformation
  - Precondition: Class does not exist



Minimalistic CT Language Family

Logic term alphabet of program / model elements (choose your own)

Σ = { class(Id,Pkg,Name), field(Id,Class, Name),
method(Id,Class,Name,...), getfield(Id,...,Meth,Field) }

Condition Language

$C \rightarrow EC \mid EC \wedge C \mid EC \vee C \mid \text{not}(C)$
$EC \rightarrow \text{true} \mid \text{false} \mid \text{exists}(\text{elem}) : \text{elem} \in \Sigma$

Transformation Language

$T \rightarrow ET \mid ET\ ,\ T$
$ET \rightarrow \text{add}(\text{elem}) \mid \text{delete}(\text{elem}) \mid \text{replace}(\text{elem}_1,\text{elem}_2) : \text{elem} \in \Sigma$

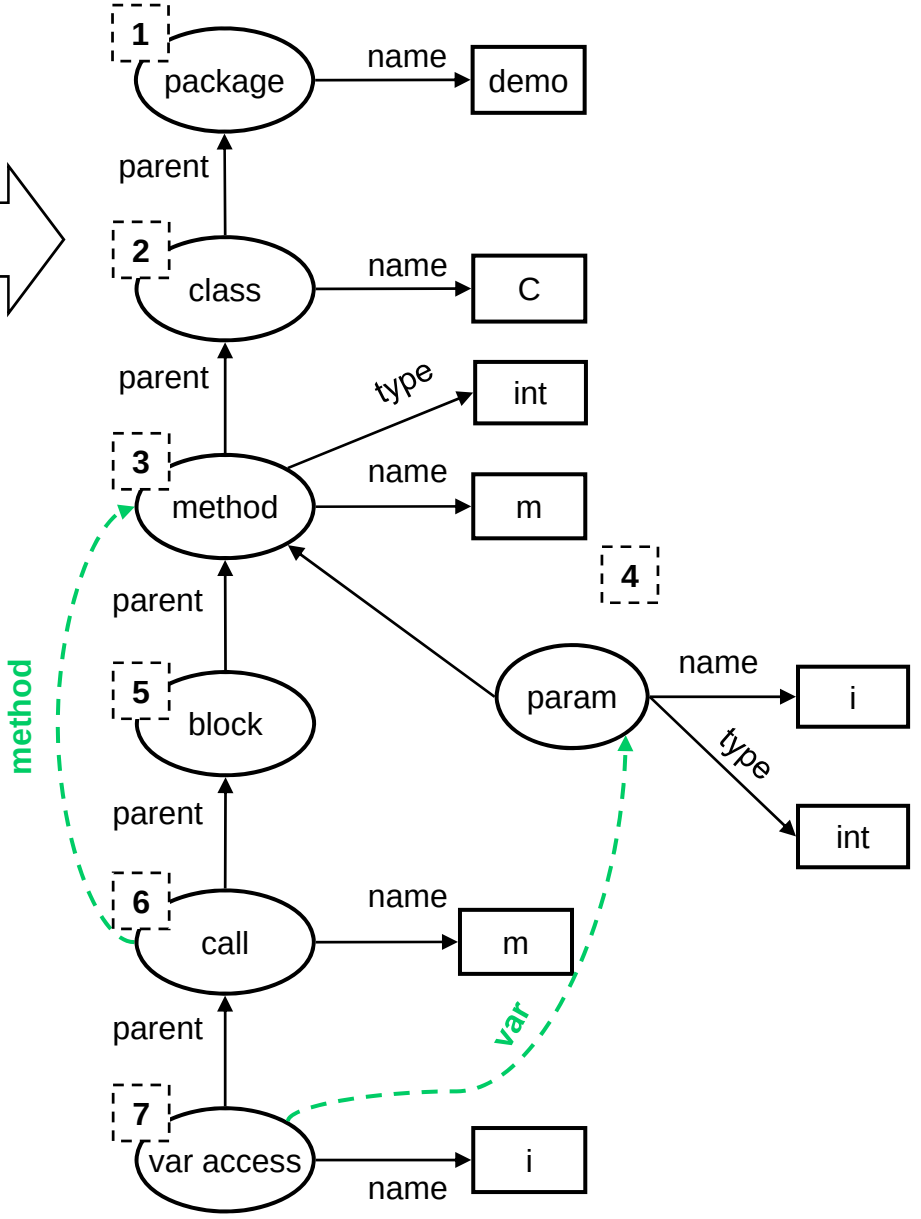
Logic Fact Representation of AST

```
package demo;
class C {
   void m( String s) {
     m(s);
   }
}
```

AST

Facts

```
package(1, 0, 'demo')
  class(2, 1, 'C')
 method(3, 2, 'm', void)
  param(4, 3, 's', 100)
  block(5, 3)
   call(6, 5, null, 3 )
  ident(7, 6, 4 )
```



Logic-based Program Analysis Example: Pattern Mining

What is the essence of a "Singleton Pattern"?

→ A static method in Type
returns an instance of Type
by accessing a static field that has type Type

```
classMethodReturnsOwnInstance(Type, Method, Field) :-

   methodDefT(Method, Type, _, [], type(_, Type, 0), _, _),
   modifierT(Method, static),

   fieldDefT(Field, Type, type(_, Type, 0),_,_),
   modifierT(Field, static),

   getFieldT(_, _, Method, _, _, Field).
```

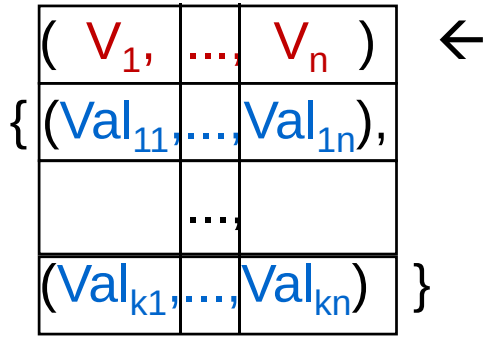
- Query: "?- classMethodReturnsOwnInstance(Type, Method, Field)."
- Returns tuples of values for <Type, Method, Field> that represent singletons.
- Generates all results via backracking.

CT are Parametric

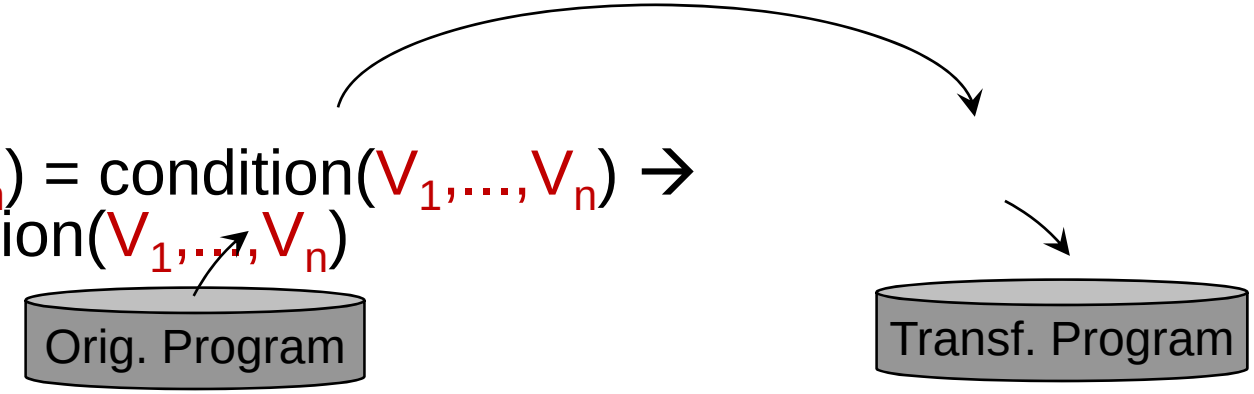
- Syntax
  - ct(head(V_1,...,V_n), condition(V_1,...,V_n), transformation(V_1,...,V_n)
- Think of it as
  - ct_head(V_1,...,V_n) := FORALL V_1,...,V_n {
  - IF condition(V_1,...,V_n) THEN transformation(V_1,...,V_n) }
- Variables describe program elements
  - to be changed
  - needed to determine that the change is legal
  - needed to express context-dependent transformations
- Application of a CT to a program
  - ∀ substitutions that make the precondition true:
    ⇒ perform the transformations subject to these substitutions

CT are Parametric

- Application of a CT to a program
  - ∀ substitutions that make the precondition true:
    ⇒ perform the transformations subject to these substitutions



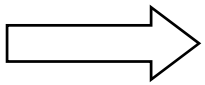
- CT(V_1,...,V_n) = condition(V_1,...,V_n) → transformation(V_1,...,V_n)



Example: Create Accessor Method

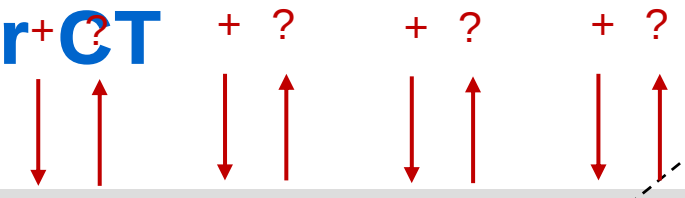
- AddGetter CT
  - for all fields that have no getter method ...
  - ... add method that returns the field's value

```
public class C {

  B b = new B();




  ...

}
```



```
public class C {

  B b = new B();

  B getB() {
     return b;
  }

   ...

}
```

AddGetterCT




```
ct( addGetter(Class,Field,Type,GName), (
    classDefT(Class,_,_,_), not(externT(Class)),
    fieldDefT(Field,Class,Type,Name,_),

    concat(get, Name, GName),
    not( methodDefT(_,Class, GName,[],Type,_,_) ),

    new_id(Method),...,new_id(Get)
),(
    add( methodDefT(Method,Class, GName,[],Type,[],Block) ),
    add( blockT(Block,Method,Method,[Return]) ),
    add( returnT(Return,Block,Method,Get) ),
    add( getFieldT(Get,Return,Method,null,Name,Field) ),
    add_to_class(Class,Method)
)).
```

No method with signature "<Type> get<Name>()" exists.

Head / Signature

Precondition

New identities for new elements

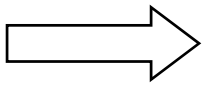
Transformation

Create method "<Type> get<Name>() { return <Field>}":

Replace Read Accesses

- ReplaceReadAccesses
  - for all fields that have a getter method and
    for all read accesses to the field outside of its getter method...
  - ... replace the read access by the getter invocation

```
public class C {

  B b = new B();

  B getB() {...}

  foo() {
     ...
     b.m();
  }
}
```



```
public class C {

  B b = new B();

  B getB() {...}

  foo() {
     ...
     getB().m();
  }
}
```

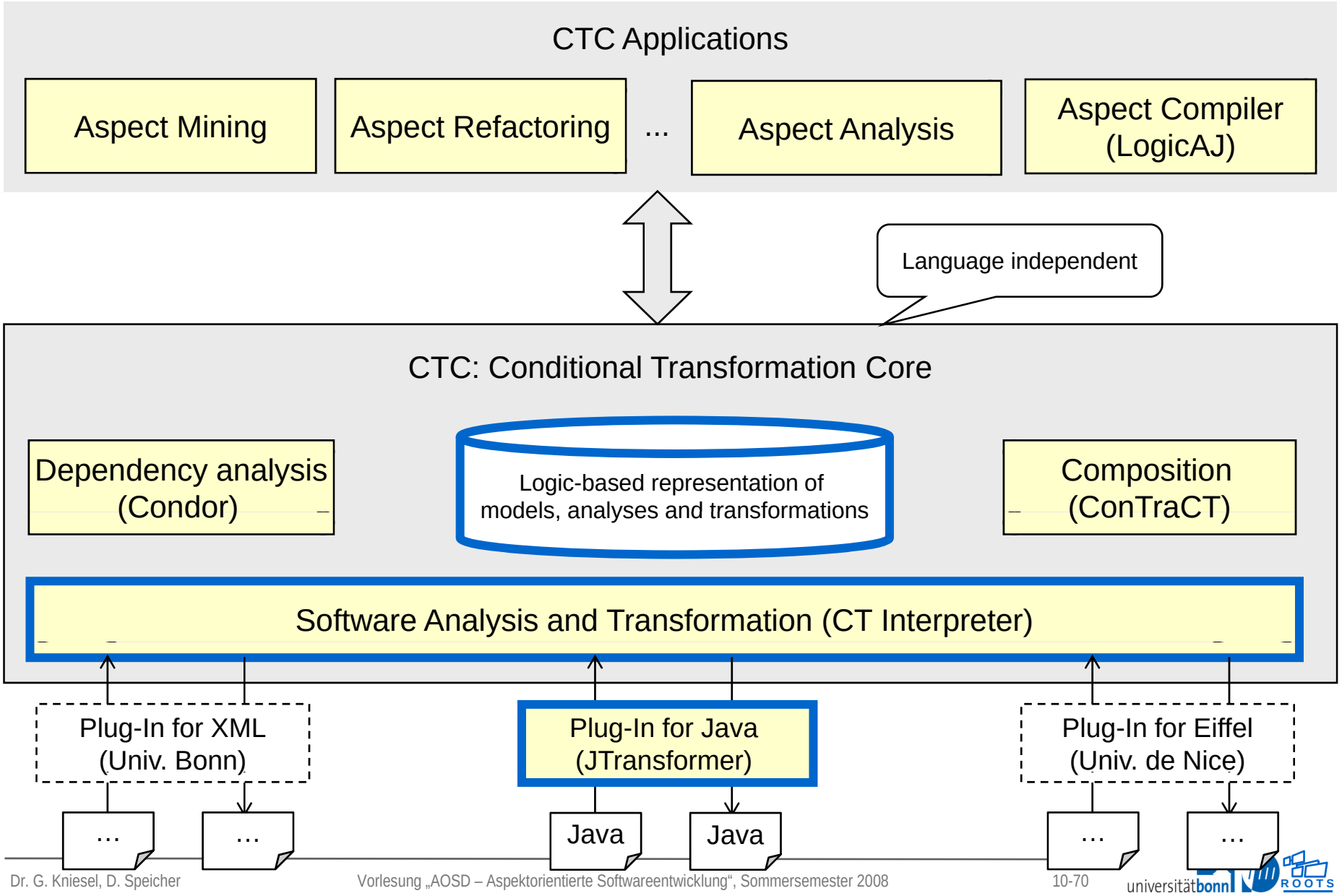
Full „Encapsulate Field“ Refactoring

- Five CT definitions
  - AddGetter
  - AddSetter
  - ReplaceReadAccesses
  - ReplaceWriteAccesses
  - MakeFieldPrivate
- A CT sequence definition
  - invoke all of the above in the specified order
- Syntax of CT invocation in JTransformer
  - apply_ct(Head): invoke a CT
  - apply_ctlist([Head1, ..., HeadN]): invoke a sequence of CTs
- Full code of Encapsulate Field example
  - http://www.cs.uni-bonn.de/~gk/XPandProgramTransformation/encapsulateField.pl

JTransformer: More infos

- JT homepage
  - http://roots.iai.uni-bonn.de/research/jtransformer
- PEF documentation
  - On homepage
- Tutorial
  - On hoomepage
- Step-by-step introduction
  - on http://www.cs.uni-bonn.de/~gk/XPandProgramTransformation/

The Big Picture: Conditional Transformations



Summary: From Aspects to CTs

Translation Scheme

- Pointcuts
  - CT precondition
- Introduction
  - CT Transformation that adds the introduced elements at the join point
- Advice
  - CT Transformation that adds the advice body at the join point
- Join Point
  - Logic variable denotes reference point of advice / introduction
    - ⇨ Field access, method call, ...
    - ⇨ Package, type, ...
  - Relative position with respect to the reference point (before / after) is implemented by updating the element order info in the program
    - ⇨ Add new statements at the proper place in the member list of the containing block
    - ⇨ Add new fields, methods and types at the proper place in the member list of the containing class

Advantages of logic-based approach

- Natural
  - Software analysis requires to express conditions → condition = logic!
- Compact
  - Complex program structures easily represented as logic terms
- Easy to use
  - Write a few predicates instead of many classes with many methods
- Thorough formal foundations
  - Enables automated analysis (→ Next lecture) and provably correct optimizations

References: CTs and their Analysis

- A Logic Foundation for Conditional Program Transformations
  - Günter Kniesel
    Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn. ISSN 0944-8535. January 2006.
- Static Composition of Refactorings
  - Günter Kniesel, Helge Koch
    In: Ralf Lämmel (Ed.): Science of Computer Programming 52 (2004), special issue on "Program Transformation", p. 9-51, Elsevier Science, 2004.
    Digital Object Identifier: doi:10.1016/j.scico.2004.03.002
    If the DOI or Wiley website is unreachable, here is a local copy of a preprint.

Next Lecture

- An Analysis of the Correctness and Completeness of Aspect Weaving
  - Günter Kniesel, Uwe Bardey
    in *Proceedings of Thirteenth Working Conference on Reverse Engineering (WCRE 2006),* Oct. 23-27, Benevento, Italy, ISBN 0-7695-2719-1, p. 324-333, IEEE 2006.
- Transformation dependency analysis - A comparison of two approaches
  - Tom Mens, Günter Kniesel, Olga Runge
    Proceedings of Langages et Modèles à Objets (LMO2006), March 22-24, 2006, Nîmes, France, special issue of L'Objet, Hermes Science Publishing, London, 2006.