Journal of Universal Computer Science, vol. 14, no. 21 (2008), 3534-3555
submitted: 16/4/08, accepted: 5/6/08, appeared: 1/12/08 © J.UCS

On the Interaction of Advices and Raw Types in AspectJ

Fernando Barden Rubbo
(Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
fbrubbo@inf.ufrgs.br)

Rodrigo Machado
(Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
rma@inf.ufrgs.br)

Álvaro Freitas Moreira
(Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
afmoreira@inf.ufrgs.br)

Leila Ribeiro
(Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
leila@inf.ufrgs.br)

Daltro José Nunes
(Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
daltro@inf.ufrgs.br)

Abstract: The latest versions of AspectJ, the most popular aspect-oriented extension for Java, must cope with the complex changes that occurred in the Java type system, specially with those that introduced type parameters for classes and methods. In this work we study the influence of *raw* types, i.e. parameterless instantiations of class types, over the semantics of an AspectJ-like language. As a result, we define an operational semantics and a type system for a calculus, named Raw Aspect Featherweight Generic Java (Raw-AFGJ), that represents a minimal aspect-oriented extension of Raw Featherweight Generic Java. Through our calculus it is possible to achieve a better understanding of several subtleties of aspect weaving with the restrictions imposed by raw types support in the type system.

Key Words: aspect-oriented programming, operational semantics, type systems
Category: D.3.1, D.3.3, F.3.2.

1 Introduction

The AspectJ language[Kiczales et al., 2001] is a superset of Java that allows the definition of aspect oriented abstractions, i.e. pointcuts, advices and aspects. Some parts of AspectJ, such as pointcut matching, rely strongly on types, making them very sensitive to modifications in the type system. As the Java language evolved, it eventually came to include – in its version 1.5 – parametric polymorphism for classes, interfaces and methods. In Java, parametric polymorphism is mainly used to provide better compile-time type checking for container classes. Naturally, such improvements raised concerns about compatibility with legacy code in both source and bytecode levels.

The source language compatibility issue was solved by expanding the type system to support parameterless instantiations of parametrized types, the so called *raw types*[Bracha et al., 1998]. With raw types, legacy classes could be transformed into polymorphic versions without affecting their monomorphic clients. For instance, in Generic Java the class `ArrayList` requires a type argument that defines its content type. Thus, objects of this class are created by calls such as `new ArrayList<Integer>()`, `new ArrayList<String>()`, and so on. In the presence of raw types, although, it is also possible to create objects with a parameterless call such as `new ArrayList()`. In this case, the type of the contents is obtained by means of *type erasure*.

Since the Java Virtual Machine has not changed substantially due to the addition of generics to the Java language, the compatibility issue in the bytecode level was solved using *type erasure*. This means that, after type checking, the Java compiler erases type parameter information in order to generate a backward-compatible bytecode.

Following Java 1.5, AspectJ has also added support for parametric polymorphism. Parameterized types may freely be used within aspects (including pointcut expressions) and support is also provided for generic abstract advices. However, although the role of raw types and type erasure in polymorphic Java is clear, the same can not be said about their role in polymorphic AspectJ.

As an example, consider the following code:

```
class Buffer<E extends Number> {

 void add_1(E e){..}
 void add_2(List e){..}
 void add_3(List<E> e){..}
 void add_4(List<Integer> e){..}

 public static void main(String[] args) {
  Buffer<Integer> b1 = new Buffer<Integer>();
  b1.add_1(42);
  b1.add_2(new ArrayList<Number>());
  b1.add_3(new ArrayList<Integer>());
  b1.add_4(new ArrayList());
 }

}
```

```
public aspect BufferAspect {

before():execution(void *(Integer)){..}

before():execution(void *(List)){..}

before():execution(void *(List<Integer>)){..}

before():execution(void *(List<Number>)){..}

}
```

The class `Buffer` represents a container whose content type depends on the given value for type parameter `E`. This class has methods that insert different kinds of data into the buffer. It also contains a `main` method that instantiates a `Buffer` object (passing `Integer` as type parameter) and adds some content to it. The aspect `BufferAspect` defines four advices that insert code before the execution of any method with exactly one parameter whose return type is `void`.

The advices differ only on the advised method parameter type: `Integer`, `List`, `List<Integer>` or `List<Number>`, respectively.

At first glance, it would be natural to think that all the advices are triggered when the `main` method executes. However, in the current `AspectJ` implementation, only the second and the third advices are weaved to the code. The first advice does not have a match because, due to type erasure, the type variable `E` in method `add_1` is replaced by its *bound* (`Number`) at compile-time. The type variable `E` is totaly erased from the type signature of `add_3`, in the bytecode. Because of that, the third advice fails to match against it. The second advice is triggered by the execution of `add_2`, `add_3` and `add_4`. This occurs because `execution` pointcuts match against the method signature of the bytecode, in this case any possible instantiation of `List`. Finally, the execution of `add_4` is advised by the third advice, since the pointcut expression defines an exact match on a type whose parameters are fully defined at compile-time.

Besides the weaving semantics, the type system of AspectJ is also affected by type erasure. For instance, consider the following advice declaration:

```
before(): execution(void Buffer<String>.foo()){
   System.out.println("Executing foo in Buffer<String>\n");
}
```

This advice aims to print a message before all executions of `foo`, but only when it is called from an object of type `Buffer<String>`. Although this seems to be a valid declaration, it is considered ill-typed by the AspectJ type system. This happens because there is no way to check the precise type parameter for `Buffer` at compile-time, and to distinguish between a `Buffer<String>` object and a `Buffer<Integer>` object. Therefore, the AspectJ type system only accepts *raw types* as valid object types in pointcut expressions.

Such subtleties involving both the weaving and the type system of AspectJ make the reasoning over aspect-oriented code somewhat misleading and difficult. Both raw types and type erasure cause an absence of type information during the execution of the program. The AspectJ language deals with this by disallowing situations where it would cause problems (e.g. unintended changes in the program execution) or simply warning the programmer during compilation. In this work, we aim to characterize the influence of *raw types* and *type erasure* in AspectJ. We define an operational semantics and a type system for a calculus, named Raw Aspect Featherweight Generic Java (Raw-AFGJ), where such influence becomes more clear. As far as we know, this is the first work that formally investigates this issue in an aspect-oriented context.

Raw-AFGJ is a combination of Raw Featherweight Generic Java [Igarashi et al., 2001b], with the aspect weaving semantics proposed in Aspect Featherweight Generic Java [Jagadeesan et al., 2006]. The main contributions of this paper are: (1) the Raw-AFGJ calculus, given by type system and operational semantics; (2) proof of the type soundness of Raw-AFGJ.

The remainder of the paper is organized as follows: in Section 2, we present our calculus, Raw-AFGJ, by means of a structural operational semantics and a type system. In Section 3, we explore the properties of the calculus, mainly type soundness. In Section 4, we present an overview of the state-of-art regarding calculi for representing Java and the AspectJ weaving semantics. Finally, in section 5 we present our conclusions.

2 Raw Aspect Featherweight Generic Java

This sections describes the Raw-AFGJ calculus by means of its abstract syntax, type system and reduction rules. The Raw-AFGJ calculus is a combination of two extensions of Featherweight Generic Java (FGJ) [Igarashi et al., 2001a], namely RawFGJ [Igarashi et al., 2001b] that provides a formal description of raw types in FGJ, and AFGJ [Jagadeesan et al., 2006], that extends FGJ with parametrized advice definitions. Since we focus on the interaction of raw types and advices, we tried to keep most of the design decisions of the original calculi. Some modifications, although, were introduced in order to bring the semantics of Raw-AFGJ closer to the one of AspectJ.

2.1 Syntax

The syntax for Raw-AFGJ programs is depicted in Figure 1. A program is a sequence of declarations followed by an expression to be executed. This expression represents the program entry point, just like the `main` method in Java. The declarations consist of a list of classes and advice declarations. In the following, we use $\overline{\texttt{e}}$ as an abbreviation for $\texttt{e}_1\texttt{,}\ \texttt{e}_2\texttt{,}\ \texttt{...,}\ \texttt{e}_n$, for $n \geq 0$. The operation $\#(\overline{\texttt{e}})$ retrieves the length n of the sequence $\overline{\texttt{e}}$. Similar conventions are used for sequences of type parameter declarations, field declaration, judgements, etc. It is important to note that a single occurrence of some metavariable `M` differs from the indexed sequence of metavariables $\overline{\texttt{M}}$.

The declaration $\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>} \triangleleft \texttt{ N \{ } \overline{\texttt{fd}}\ \texttt{k}\ \overline{\texttt{md}}\ \texttt{\}}$ defines a class `C` with a list $\overline{\texttt{X}\triangleleft\texttt{N}}$ of type parameters, a parent type `N`, a list $\overline{\texttt{fd}}$ of field definitions, a constructor definition `k`, and a list $\overline{\texttt{md}}$ of method definitions. The symbol $\triangleleft$ substitutes the `extends` keyword for the sake of brevity. A type parameter definition $\texttt{X}\triangleleft\texttt{N}$ states that the type variable `X` will be constrained to a subtype of the non-variable type `N`. The `Object` class is implicitly defined, i.e. it can not occur as the name of a declared class. Every declared class must specify a superclass, even when the superclass is `Object`. A field declaration `T f` simply associates type `T` to a field name `f`. Moreover, field definitions can not be overwritten by subclasses, i.e. inheritance is conservative for fields (but not for methods). The constructor definition $\texttt{C (}\overline{\texttt{T g}}\texttt{, }\overline{\texttt{T f}}\texttt{) \{ super(}\overline{\texttt{g}}\texttt{) }\overline{\texttt{this.f=f}}\texttt{ \}}$ is the only mandatory part of the class body. The constructor parameters $\overline{\texttt{T g}}$ are used to initialize object

Metavariables:

C,D,E	classes	N,P,Q	non-var types
f,g	fields	a,b	advices
m	methods	ϕ,ψ,ρ	pointcuts
d,e	expressions	X,Y,Z	type variables
v,u,w	values	x,y,z	variables
R,S,T,U,V	types		

Class Definition:

cd ::= class C< $\overline{\texttt{X}\triangleleft\texttt{N}}$ > $\triangleleft$ N { $\overline{\texttt{fd}}$ k $\overline{\texttt{md}}$ }	class
fd ::= T f	field
k ::= C($\overline{\texttt{T g}}$, $\overline{\texttt{T f}}$){super($\overline{\texttt{g}}$) $\overline{\texttt{this.f=f}}$ }	constr.
md ::= < $\overline{\texttt{X}\triangleleft\texttt{N}}$ > T m ($\overline{\texttt{T x}}$) { e }	method

Advice Definition:

ad ::= advice a T ($\overline{\texttt{T x}}$) : ϕ { e }

Pointcuts:

ϕ ::= exe T C.m($\overline{\texttt{T}}$)
  | exe T C.*($\overline{\texttt{T}}$)
  | exe T C.m($\overline{\texttt{T}}$,*)
  | exe T C.*($\overline{\texttt{T}}$,*)
  | ϕ && ψ
  | ϕ || ψ
  | false
  | true

Program:

prog ::= $\overline{\texttt{cd}}$ $\overline{\texttt{ad}}$ e

Types:

T ::= X	type variable
| C< $\overline{\texttt{T}}$ >	cooked type
| C	raw type
| $\star$	bottom type

Non-variable types:

N ::= C< $\overline{\texttt{T}}$ >	cooked type
| C	raw type

Expressions:

e ::= x	variables, this
| new N($\overline{\texttt{e}}$)	object instantiation
| e.f	fied access
| e.m< $\overline{\texttt{T}}$ >($\overline{\texttt{e}}$)	method call
| e.m($\overline{\texttt{e}}$)	raw method call
| e.m[$\overline{\texttt{a}}$]< $\overline{\texttt{T}}$ >($\overline{\texttt{e}}$)	advised method call
| e.m[$\overline{\texttt{a}}$]($\overline{\texttt{e}}$)	advised raw method call
| proceed($\overline{\texttt{e}}$)	proceed call

Values:

v ::= new N($\overline{\texttt{v}}$)	value

Figure 1: Raw-AFGJ : Abstract Syntax

fields defined in the superclass by means of super($\overline{\texttt{g}}$), while the parameters $\overline{\texttt{T f}}$ are used to initialize the locally defined fields through $\overline{\texttt{this.f=f}}$. A method declaration < $\overline{\texttt{X}\triangleleft\texttt{N}}$ > T m ($\overline{\texttt{T x}}$) { e } defines type parameters $\overline{\texttt{X}\triangleleft\texttt{T}}$, a return type T, a method name m, method parameters $\overline{\texttt{x}}$ and a method body e.

An expression can be a variable reference x, a constructor invocation new N($\overline{\texttt{e}}$), a field access e.f, a parametrized method call e.m< $\overline{\texttt{T}}$ >($\overline{\texttt{e}}$), a raw method call e.m($\overline{\texttt{e}}$), or advised versions of the two kinds of method invocation (e.m[$\overline{\texttt{a}}$]< $\overline{\texttt{T}}$ >($\overline{\texttt{e}}$) and e.m[$\overline{\texttt{a}}$]($\overline{\texttt{e}}$), respectively). The advised methods are equipped with a list [$\overline{\texttt{a}}$] of names of advices which remain to be run before the body execution, representing the weaving. Note that the advised method calls are not part of the source language, occurring only as intermediate steps of the method evaluation.

The declaration advice a T ($\overline{\texttt{T x}}$) : ϕ {e} defines and advice of name a. The signature T ($\overline{\texttt{T x}}$) represents the advice type, i.e. the type of the piece of code that will execute in place of some method calls. The pointcut expression ϕ represents what methods should be intercepted by the advice. The behavior of advices in Raw-AFGJ is similar to the `around` advice in AspectJ, executing the advice body `e` if the pointcut ϕ is matched. Inside the advice body it is possible to call the special method `proceed`, which is an alias for the method currently being advised.

In Raw-AFGJ we model essentially method execution join points. We use the syntax `exe T C.m(`$\overline{\texttt{T}}$`)` in the same sense as the `execution` pointcut of AspectJ, matching a method within class `C`, with name `m`, with parameters whose types are $\overline{\texttt{T}}$, that returns a value of type `T`. In some places, like in method name or in type parameters, we can have the wildcard replacement `*`, meaning any match. We can also have as pointcuts atomic boolean values (`true` and `false`) and combinations of pointcuts using the `or` and `and` operators.

2.2 Type System

The language of types of Raw-AFGJ, depicted in Figure 1, is basically the same as in Raw FGJ. Types in general are represented by the meta-variable `T`, and we use `N` to range over non-variable types. Every class definition has a (possibly empty) list of type variables. So, every reference to a class can be either *cooked* (when the type parameters are defined) or *raw* (without type parameters). A parametrized type of a class `C`, denoted as `C<`$\overline{\texttt{T}}$`>`, is different from the correspondent raw type, denoted by `C` alone, even if the list of type parameters is empty (`C<>`). The bottom type $\star$ is used as a placeholder for unknown type parameters. This is needed to provide valid types for both raw methods and constructors of raw classes. Type variables, denoted by `X`, `Y` and `Z`, can occur in type expressions as well.

A *type variable environment*, denote by Δ, is a (possibly empty) sequence of associations `X` $<:$ `N`, representing that the type variable `X` is bounded by the non-variable type `N`. The association of variables to types is given by a *type environment*, denoted by Γ. A type environment can also have an special form of association, `T proceed(`$\overline{\texttt{T}}$`)`, which is needed by the type system. It is used to save the type of the weaved methods when typing the advice body.

2.2.1 Well-Formed Types and Environments

The judgement $\Delta \vdash$ `T` ok denotes that the type `T` is well-formed with relation to the type variable environment Δ. It means that all type variables that occur in `T` must respect the bounds defined by the type variable environment Δ. Moreover, well-formed class types must also be valid with respect to the type parameter restrictions provided by the class definitions. The rules for well-formed type judgements are shown below.

$$\Delta \vdash \texttt{Object}\ \text{ok} \qquad (\text{WF-OBJECT})$$

$$\Delta \vdash \star\ \text{ok} \qquad (\text{WF-BOT})$$

$$\frac{\texttt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \texttt{X}\ \text{ok}} \qquad (\text{WF-VAR})$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\lhd\texttt{N}}\texttt{>} \lhd \texttt{N \{ ... \}} \qquad \Delta \vdash \overline{\texttt{T}}\ \text{ok} \qquad \Delta \vdash \overline{\texttt{T}} <: [\overline{\texttt{T}/\texttt{X}}]\overline{\texttt{N}}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ \text{ok}} \qquad (\text{WF-CLASS})$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\lhd\texttt{N}}\texttt{>} \lhd \texttt{N \{ ... \}}}{\Delta \vdash \texttt{C}\ \text{ok}} \qquad (\text{WF-RAW})$$

The syntax $[\overline{\texttt{T/X}}]\texttt{N}$ means the application of a sequence of type variable substitutions $\texttt{T}_i/\texttt{X}_i$ in N.

A type variable environment Δ is said to be well-formed, written $\Delta \vdash$ ok if all of its bounds are well-formed types under itself, i.e. if for all $\texttt{X}<:\texttt{N}$ in Δ, we have that $\Delta \vdash \texttt{N}$ ok.

A type environment Γ is said to be well-formed with respect to a type variable environment Δ, written $\Delta;\Gamma \vdash$ ok, if for all $\texttt{T x} \in \Gamma$, we have $\Delta \vdash \texttt{T}$ ok. Also, for the special form $\texttt{T proceed(}\overline{\texttt{T}}\texttt{)}$, we have $\Delta \vdash \texttt{T},\overline{\texttt{T}}$ ok.

2.2.2 Subtype relation

The judgement $\Delta \vdash \texttt{T} <: \texttt{V}$ says that type T is a subtype of V under type environment Δ. The subtype relation <: is induced by the class hierarchy. Since a class type parameter can occur both in the return type and in the parameter types of methods, the subtype relation must be invariant for type parameters. In other words, $\texttt{T} <: \texttt{S} \not\Rightarrow \texttt{C<T>} <: \texttt{C<S>}$. Considering a class definition like `class C<X◁Object> ◁ N { ... }` the raw type C is the supertype of all its possible parametrizations, like `C<Boolean>`, `C<String>` or `C<⋆>`. The type $\star$ is considered to be the minimum element of the subtype hierarchy. It is important to note that parametrizations with bottom type are not subtypes of other instantiated types, i.e. $\texttt{C<}\star\texttt{>} \not<: \texttt{C<String>}$. The rules that define the subtype relation are shown as follows.

$$\Delta \vdash \texttt{T} <: \texttt{T} \qquad (\text{S-REFLEX})$$

$$\frac{\Delta \vdash \texttt{S} <: \texttt{T} \qquad \Delta \vdash \texttt{T} <: \texttt{U}}{\Delta \vdash \texttt{S} <: \texttt{U}} \qquad (\text{S-TRANS})$$

$$\Delta \vdash \texttt{X} <: \Delta(\texttt{X}) \qquad (\text{S-VAR})$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\lhd\texttt{N}}\texttt{>} \lhd \texttt{D<}\overline{\texttt{S}}\texttt{> \{ ... \}}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <: [\overline{\texttt{T/X}}]\texttt{D<}\overline{\texttt{S}}\texttt{>}} \qquad (\text{S-SUPER})$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\lhd\texttt{N}}\texttt{>} \lhd \texttt{D \{ ... \}} \qquad \texttt{class D<}\overline{\texttt{Y}\lhd\texttt{P}}\texttt{>} \lhd \texttt{N \{ ... \}} \qquad \#(\overline{\texttt{Y}}) = \#(\overline{\star})}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{D<}\overline{\star}\texttt{>}} \qquad (\text{S-RAWSUPER})$$

$$\Delta \vdash \star <: \texttt{T} \qquad (\text{S-BOT})$$

$$\frac{\texttt{C} \trianglelefteq \texttt{D}}{\Delta \vdash \texttt{C} <: \texttt{D}} \qquad (\text{S-RAW})$$

$$\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{C} \qquad (\text{S-COOKEDRAW})$$

The relation $\trianglelefteq$, used by rule S-RAW, is the reflexive and transitive closure of the *extends* relation $\lhd$ between classes. The rule S-COOKEDRAW says that the cooked type `C<Int>` is subtype of the raw type C. However, in order to provide compatibility with old code that instantiates classes without type parameters, we may also wish that raw types were used in place of cooked types. This operation should also provide some kind of warning that such operation is not safe. The relation that includes safe subtyping but also allows such coercion is called the *unsafe subtyping relation*, and is denoted as $<:_?$. The rules for unsafe type judgements are provided as follows.

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T}}{\Delta \vdash \mathtt{S} <:_? \mathtt{T}} \quad \text{(SU-Safe)} \qquad \Bigg| \qquad \frac{\Delta \vdash \mathtt{C} <: \mathtt{D} \qquad \textit{warning}}{\Delta \vdash \mathtt{C} <:_? \mathtt{D<}\overline{\mathtt{T}}\mathtt{>}} \quad \text{(SU-Unsafe)}$$

2.2.3 Class Declaration typing

Class declarations are valid if the type variable bounds, superclass type and field types are all well-typed. Moreover, the constructor arguments that are passed by the `super` call must coincide with the constructor arguments for the supertype class. Finally, all declared methods must also be valid inside the class. The following rule checks such constraints for class definitions. The auxiliar function *cargtype* retrieves the argument types for the class constructor, triggering a *warning* if the types are changed due to type erasure (for more details, see [Igarashi et al., 2001b]).

$$\frac{\begin{array}{c}\overline{\mathtt{X<:N}} \vdash \overline{\mathtt{N}},\overline{\mathtt{T}},\mathtt{N}\ \text{ok} \qquad \textit{cargtype}(\mathtt{N}) = \overline{\mathtt{U\ g}} \qquad \overline{\mathtt{md}}\ \text{ok}\ \mathtt{in\ C<}\overline{\mathtt{X\triangleleft N}}\mathtt{>} \\ \mathtt{k\ =\ C\ (}\overline{\mathtt{U\ g}}\mathtt{,}\overline{\mathtt{T\ f}}\mathtt{)\ \{\ super(}\overline{\mathtt{g}}\mathtt{)}\ \overline{\mathtt{this.f=f}}\ \mathtt{\}}\end{array}}{\mathtt{class\ C<}\overline{\mathtt{X\triangleleft N}}\mathtt{>\ \triangleleft\ N\ \{\ }\overline{\mathtt{T\ f}}\ \mathtt{k}\ \overline{\mathtt{md}}\ \mathtt{\}}\ \text{ok}} \quad \text{(T-Class)}$$

Method overloading is not allowed, i.e. in a given class `C` the same method name `m` can not have two definitions with different type parameters. This choise eases method lookup, since it does not have to match parameter types. In FJ, methods can be overridden in subclasses only if they have the same type as their ascendent. In Raw-AFGJ (which is based on FGJ), covariance of the return type in overriding is also allowed. The following rules are used to determine if a given method `m`, with type $\mathtt{<}\overline{\mathtt{Y\triangleleft P}}\mathtt{>}\overline{\mathtt{T}}\to\mathtt{T}$, is a valid overring of method in superclass `N`. If the method is not defined in `N`, it is trivially valid. Otherwise, the rules verify if the types are compatible.

$$\frac{\textit{mtype}(\mathtt{m},\mathtt{N}) = \mathtt{<}\overline{\mathtt{Z\triangleleft Q}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}\ \text{implies}\ \ \overline{\mathtt{P}} = [\overline{\mathtt{Y/Z}}]\overline{\mathtt{Q}}\ \ \text{and}\ \overline{\mathtt{T}} = [\overline{\mathtt{Y/Z}}]\overline{\mathtt{U}}\ \text{and}\ \Delta \vdash \mathtt{T} <: [\overline{\mathtt{Y/Z}}]\mathtt{U}}{\textit{override}(\mathtt{m},\mathtt{<}\overline{\mathtt{Y\triangleleft P}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T},\mathtt{N})} \quad \text{(Override)}$$

$$\frac{\textit{mtyperaw}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{U}} \to \mathtt{U}\ \text{implies}\ \ \overline{\mathtt{U}} = \overline{\mathtt{T}}\ \text{and}\ \Delta \vdash \mathtt{T} <: \mathtt{U}}{\textit{override}(\mathtt{m},\overline{\mathtt{T}} \to \mathtt{T},\mathtt{C})} \quad \text{(OverrideR)}$$

Method definition typing must check if the type variable bounds, parameter and return types are well-defined. The method body type must be consistent with the declared return type, considering the type environment augmented with types for parameters and the special variable `this`. Moreover, the method definition must also be a valid override.

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X<:N}},\overline{\mathtt{Y<:P}} \qquad \mathtt{class\ C<}\overline{\mathtt{X\triangleleft N}}\mathtt{>\ \triangleleft\ N\ \{\ }\overline{\mathtt{fd}}\ \mathtt{k}\ \overline{\mathtt{md}}\ \mathtt{\}} \\ \Delta \vdash \overline{\mathtt{T}},\mathtt{T},\overline{\mathtt{P}}\ \text{ok} \qquad \Delta;\overline{\mathtt{T\ x}},\mathtt{V\ this} \vdash \mathtt{e} : \mathtt{S} \qquad \Delta \vdash \mathtt{S} <: \mathtt{T} \\ \textit{override}(\mathtt{m},\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T},\mathtt{N})\end{array}}{\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>T\ m(}\overline{\mathtt{T\ x}}\mathtt{)\{\ e\ \}}\ \text{ok}\ \mathtt{in\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}} \quad \text{(T-Method)}$$

2.2.4 Advice Declaration typing

Advice definitions must take into account if the types of the parameters, return type and target class are all valid in the empty type variable environment (since they do not have type variables). The type of the aspect body must be valid under the type environment extended with the types for `target` (an alias for the class of the advised method) and `proceed` (an alias for the advised method itself). The pointcut logic relation $\models$ (described in Section 2.2.5) is used to provide the good formation of the pointcut definition. In AFGJ, type variables where allowed in advices. We removed parametrization in advices since AspectJ only provides weaving for concrete advices.

$$\frac{\begin{array}{c}\phi \models \texttt{exe T C.*}(\overline{\texttt{T}}\texttt{,*}) \qquad \varnothing \vdash \texttt{T},\overline{\texttt{T}},\texttt{C ok} \\ \varnothing;\overline{\texttt{T x}},\texttt{C target},\texttt{T proceed}(\overline{\texttt{T}}) \vdash \texttt{e} : \texttt{T'} \qquad \varnothing \vdash \texttt{T'} <: \texttt{T}\end{array}}{\texttt{advice a T }(\overline{\texttt{T x}}) : \phi \texttt{ \{ e \} ok}} \quad \text{(T-ADV)}$$

The advised method calls contain a list of advice names that intercept the method execution. The weaving of advices in methods are represented by the relation *advisedby* (defined below) in which `C.m` *advisedby* `a` means that the advice `a` intercepts method `m` in class `C`.

$$\frac{\begin{array}{c}\texttt{advice a T }(\overline{\texttt{T x}}) : \phi \texttt{ \{ e \}} \qquad \texttt{exe T C.m }(\overline{\texttt{T}}) \models \phi \\ \mathit{mtypeadv}(\texttt{m},\texttt{C}) = \overline{\texttt{T}_0} \to \texttt{T}_0 \qquad \mathit{matcheq}(\overline{\texttt{T}_0},\overline{\texttt{T}}) \qquad \mathit{matcheq}(\texttt{T}_0,\texttt{T})\end{array}}{\texttt{C.m}\ \mathit{advisedby}\ \texttt{a}} \quad \text{(ADVISED-BY)}$$

The *advisedby* is used by both type system and reduction semantics and it is based on $\models$ relation for matching pointcuts against method calls. Since this relation forces all types to match exactly, we modified the matching process in order to include the treatment for *raw* types.

1. If there are type variables in any type definition on the target method signature we must perform the erasure of these types before matching any pointcut against it. We represent this conditional signature erasure of type `T` under type variable environment Δ as $\|\texttt{T}\|_\Delta$. This function is used by the method type fetching function *mtypeadv*. Both functions are defined as follows:

$$\frac{\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>} \triangleleft \texttt{N \{ }\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{md}}}{\mathit{mtypeadv}(\texttt{m},\texttt{C}) = \mathit{mtypeadv}(\texttt{m},\mathit{head}(\texttt{N}))}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>} \triangleleft \texttt{N \{ }\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{ \}} \\ \texttt{<}\overline{\texttt{Y}\triangleleft\texttt{P}}\texttt{> R m }(\overline{\texttt{U x}})\texttt{ \{ e \}} \in \overline{\texttt{md}} \qquad \Delta = \overline{\texttt{X}<:\texttt{N}},\overline{\texttt{Y}<:\texttt{P}}\end{array}}{\mathit{mtypeadv}(\texttt{m},\texttt{C}) = \|\overline{\texttt{U}}\|_\Delta \to \|\texttt{R}\|_\Delta}$$

$$\|\texttt{T}\|_\Delta = \begin{cases} \texttt{T} & \text{if } \varnothing \vdash \texttt{T ok} \\ |\texttt{T}|_\Delta & \text{otherwise} \end{cases}$$

2. If the pointcut return or if the pointcut parameters are defined as raw types we must match it against erasured method signatures. For example, if the pointcut is `exe List C.*()`, we must pick up all methods defined in class `C` whose the erasure of return type is `List` (e.g. `List`, `List<Int<>>`, `List<X>` and others). However, if the poitcut is using a fully defined type,

`exe List<Int<>> C.*()` for example, we must perform an exact match (i.e. only methods with `List<Int<>>` as its return type must be picked up). To ensure this functionality we have created another auxiliary function, named *matcheq*.

$$matcheq(T_0, T)\begin{cases} head(\mathtt{T}_0) = \mathtt{T} & \text{if T is a raw type} \\ \mathtt{T}_0 = \mathtt{T} & \text{otherwise} \end{cases}$$

2.2.5 Pointcut logic

The matching of execution points and advices is determined by a monotonic logic, initially proposed in [Jagadeesan et al., 2006]. The rules that define the relation $\models$ are as follows.

$$\rho \models \mathtt{true} \qquad \mathtt{false} \models \rho \qquad \frac{\psi \models \rho}{\phi \;\&\&\; \psi \models \rho} \qquad \frac{\rho \models \phi \quad \rho \models \psi}{\rho \models \phi \;\&\&\; \psi}$$

$$\frac{\phi \models \rho}{\phi \;\&\&\; \psi \models \rho} \qquad \frac{\rho \models \phi}{\rho \models \phi \;||\; \psi} \qquad \frac{\rho \models \psi}{\rho \models \phi \;||\; \psi} \qquad \frac{\psi \models \rho \quad \phi \models \rho}{\phi \;||\; \psi \models \rho}$$

$$\begin{array}{ll} \mathtt{exe\ T\ C.m}(\overline{\mathtt{T}},\overline{\mathtt{S}}) \models \mathtt{exe\ T\ C.m}(\overline{\mathtt{T}},*) & \mathtt{exe\ T\ C.m}(\overline{\mathtt{T}},\overline{\mathtt{S}},*) \models \mathtt{exe\ T\ C.*}(\overline{\mathtt{T}},*) \\ \mathtt{exe\ T\ C.m}(\overline{\mathtt{T}},\overline{\mathtt{S}},*) \models \mathtt{exe\ T\ C.m}(\overline{\mathtt{T}},*) & \mathtt{exe\ T\ C.*}(\overline{\mathtt{T}},\overline{\mathtt{S}}) \models \mathtt{exe\ T\ C.*}(\overline{\mathtt{T}},*) \\ \mathtt{exe\ T\ C.m}(\overline{\mathtt{T}}) \models \mathtt{exe\ T\ C.*}(\overline{\mathtt{T}}) & \mathtt{exe\ T\ C.*}(\overline{\mathtt{T}},\overline{\mathtt{S}},*) \models \mathtt{exe\ T\ C.*}(\overline{\mathtt{T}},*) \\ \mathtt{exe\ T\ C.m}(\overline{\mathtt{T}},\overline{\mathtt{S}}) \models \mathtt{exe\ T\ C.*}(\overline{\mathtt{T}},*) & \end{array}$$

The relation $\models$ defines how the program join points are matched by pointcuts, dealing with both the instantiation of wildcards in pointcut expressions and the combination of matchings. This logic allows to match methods with any number of parameters by means of the wildcard $*$. Roughly speaking, the pointcut ϕ is less general (or more unified) than the pointcut ψ if $\phi \models \psi$. The pointcut logic is used in the type system, when verifing the good formation of pointcuts in advices (rule T-ADVICE), and in the reduction semantics, when providing the matches for the generation of advice lists (rules R-WEAVE and R-WEAVER).

2.2.6 Expression typing

For both type system and operational semantics, the sequence $\overline{\mathtt{cd}}$ of class definitions of the program is treated as a global constant (and therefore omitted from deduction rules). Whenever a class definition $\mathtt{class\ C}\texttt{<}\overline{\mathtt{X}\triangleleft\mathtt{T}}\texttt{>}\triangleleft\mathtt{N}\ \ldots$ appears as a precondition of a rule, it is implied that `C` is defined in $\overline{\mathtt{cd}}$. The class list $\overline{\mathtt{cd}}$ must obbey some sanity conditions in order to be well-formed:

- The special class `Object` must not occur in the class table, i.e. $\mathtt{Object} \notin \overline{\mathtt{cd}}$. The rules that deal with field and method lookup have special forms to deal with this class, since `Object` has no fields and no methods.
- The class hierarchy must be acyclic.

– No class can redefine fields already defined in some superclass.

– There is at most one definition for each class $\mathtt{C} \in \overline{\mathtt{cd}}$.

– The type $\star$ does not occur anywhere in class or method definitions.

– For all $\mathtt{cd}_i \in \overline{\mathtt{cd}}$, $\mathtt{cd}_i$ ok.

The sequence of advice declarations $\overline{\mathtt{ad}}$ works in a similar fashion as the class definition list. It is also considered an implicit constant in deduction rules, and it must also respect the sanity conditions that advice names are unique, $\star$ does not occur in advice declarations and for all $\mathtt{ad}_i \in \overline{\mathtt{ad}}$, $\mathtt{ad}_i$ ok. The judgement $\overline{\mathtt{ad}};\overline{\mathtt{cd}} \vdash$ ok states that both the class declarations and advice declarations are well-formed. One relevant difference between the advice list and the class list is that the occurrence position in $\overline{\mathtt{ad}}$ is relevant. The order in which advices are declared implies the order in which they are weaved to methods, defining a priority between advices.

Type judgements of the form $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ provide typing for expressions. The rules for expression typing are shown in Figure 2. The subtleties worth noticing in expression typing are mainly related to raw and advised method calls. When a method invocation lacks type parameters, the rules exchange the unknown types for $\star$. The rule for typing advised method call check if all advice names in the list $\overline{\mathtt{a}}$ are related to it by means of the *advisedby* relation.

2.3 Operational Semantics

The execution of Raw-AFGJ programs is given by means of standard structural operational semantics. The reduction judgement $\Delta;\Gamma \vdash \mathtt{e}_1 \longrightarrow \mathtt{e}_2$ tells that program $\overline{\mathtt{cd}}\ \overline{\mathtt{ad}}\ \mathtt{e}_1$ reduces to $\overline{\mathtt{cd}}\ \overline{\mathtt{ad}}\ \mathtt{e}_2$ under environments Δ and Γ. We consider $\overline{\mathtt{cd}}$ and $\overline{\mathtt{ad}}$ to be constants, thus reducing the notational burden.

The evaluation of `e.f` by rule R-Field is actually a selection over the arguments of the constructor. The function *fields* provides the index of `f`.

The main influence of advice weaving occurs in the method evaluation. In Raw-AFGJ, a non-advised method is not directly executed. Before performing the method substitution, a list of incident advices is generated according to the *advisedby* relation and the advice definition list $\overline{\mathtt{ad}}$. The advices are triggered in the same order in which they are declared. Thus, a simple (non-advised) method call is rewritten into an advised call with the advices that will intercept the method, as shown by the rules R-Weave and R-WeaveR.

Concerning the evaluation of advised method calls, the main idea is to substitute the expression with the body of the first advice of the list, passing the method call itself (without the head of the advice list) as the value of the `proceed` statement. This is done by the rules R-Advice and R-AdviceR.

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash \text{ok} \qquad \Delta\vdash \mathtt{C<}\overline{\mathtt{U}}\mathtt{>}\ \text{ok} \qquad \Delta\vdash\overline{\mathtt{S}<:_?\mathtt{T}}\\ \mathit{cargtype}(\mathtt{C<}\overline{\mathtt{U}}\mathtt{>})=\overline{\mathtt{T}\ \mathtt{f}} \qquad \Delta;\Gamma\vdash\overline{\mathtt{e}:\mathtt{S}}\end{array}}{\Delta;\Gamma\vdash \mathtt{new\ C<}\overline{\mathtt{U}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}:\mathtt{C<}\overline{\mathtt{U}}\mathtt{>}}\ (\text{T-New})$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash \text{ok} \qquad \Delta\vdash \mathtt{C<}\overline{\star}\mathtt{>}\ \text{ok} \qquad \Delta\vdash\overline{\mathtt{S}<:_?\mathtt{T}}\\ \mathit{cargtype}(\mathtt{C})=\overline{\mathtt{T}\ \mathtt{f}} \qquad \Delta;\Gamma\vdash\overline{\mathtt{e}:\mathtt{S}}\end{array}}{\Delta;\Gamma\vdash \mathtt{new\ C}(\overline{\mathtt{e}}):\mathtt{C<}\overline{\star}\mathtt{>}}\ (\text{T-NewR})$$

$$\frac{\Delta;\Gamma\vdash \text{ok} \qquad \Gamma(\mathtt{x})=\mathtt{T}}{\Delta;\Gamma\vdash\mathtt{x}:\mathtt{T}}\ (\text{T-Var})$$

$$\frac{\Delta;\Gamma\vdash\mathtt{e}:\mathtt{T} \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}))=\overline{\mathtt{T}\ \mathtt{f}}}{\Delta;\Gamma\vdash\mathtt{e.f}_i:\mathtt{T}_i}\ (\text{T-Field})$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash\mathtt{e}:\mathtt{T} \qquad \Delta\vdash\overline{\mathtt{V}\ \text{ok}}\\ \mathit{mtype}(\mathtt{m},\mathit{bound}_\Delta(\mathtt{T}))=\mathtt{<}\overline{\mathtt{Y}\lhd\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}}\rightarrow\mathtt{R}\\ \Delta\vdash\overline{\mathtt{V}<:[\overline{\mathtt{V}/\mathtt{Y}}]\mathtt{P}} \qquad \Delta;\Gamma\vdash\overline{\mathtt{d}:\mathtt{S}} \qquad \Delta\vdash\overline{\mathtt{S}<:_?[\overline{\mathtt{V}/\mathtt{Y}}]\mathtt{U}}\end{array}}{\Delta;\Gamma\vdash\mathtt{e.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{d}}\mathtt{)}:[\overline{\mathtt{V}/\mathtt{Y}}]\mathtt{R}}\ (\text{T-Invk})$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash\mathtt{e}:\mathtt{T} \qquad \Delta\vdash\overline{\mathtt{S}<:_?\mathtt{U}}\\ \mathit{mtyperaw}(\mathtt{m},|\mathtt{T}|_\Delta)=\overline{\mathtt{U}}\rightarrow\mathtt{R} \qquad \Delta;\Gamma\vdash\overline{\mathtt{d}:\mathtt{S}}\end{array}}{\Delta;\Gamma\vdash\mathtt{e.m(}\overline{\mathtt{d}}\mathtt{)}:\mathtt{R}}\ (\text{T-InvkR})$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash\mathtt{e}:\mathtt{T} \qquad \Delta\vdash\overline{\mathtt{V}\ \text{ok}}\\ \mathit{mtype}(\mathtt{m},\mathit{bound}_\Delta(\mathtt{T}))=\mathtt{<}\overline{\mathtt{Y}\lhd\mathtt{P}}\mathtt{>}\ \overline{\mathtt{U}}\rightarrow\mathtt{R}\\ \Delta\vdash\overline{\mathtt{V}<:[\overline{\mathtt{V}/\mathtt{Y}}]\mathtt{P}} \qquad \Delta;\Gamma\vdash\overline{\mathtt{d}:\mathtt{S}} \qquad \Delta\vdash\overline{\mathtt{S}<:_?[\overline{\mathtt{V}/\mathtt{Y}}]\mathtt{U}}\\ \forall\mathtt{a}_i\in\overline{\mathtt{a}},\ \mathtt{T.m}\ \mathit{advisedby}\ \mathtt{a}_i\end{array}}{\Delta;\Gamma\vdash\mathtt{e.m<}\overline{\mathtt{V}}\mathtt{>[}\overline{\mathtt{a}}\mathtt{]\,(}\overline{\mathtt{d}}\mathtt{)}:[\overline{\mathtt{V}/\mathtt{Y}}]\mathtt{R}}\ (\text{T-InvkA})$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash\mathtt{e}:\mathtt{T} \qquad \Delta\vdash\overline{\mathtt{S}<:_?\mathtt{U}}\\ \mathit{mtyperaw}(\mathtt{m},|\mathtt{T}|_\Delta)=\overline{\mathtt{U}}\rightarrow\mathtt{R} \qquad \Delta;\Gamma\vdash\overline{\mathtt{d}:\mathtt{S}}\\ \forall\mathtt{a}_i\in\overline{\mathtt{a}},\ \mathtt{T.m}\ \mathit{advisedby}\ \mathtt{a}_i\end{array}}{\Delta;\Gamma\vdash\mathtt{e.m[}\overline{\mathtt{a}}\mathtt{]\,(}\overline{\mathtt{d}}\mathtt{)}:\mathtt{R}}\ (\text{T-InvkRA})$$

$$\frac{\Delta;\Gamma\vdash\mathtt{e}:\star \qquad \mathit{warning}}{\Delta;\Gamma\vdash\mathtt{e.f}_i:\star}\ (\text{T-Bot-Field})$$

$$\frac{\Delta;\Gamma\vdash\mathtt{e}:\star \qquad \Delta\vdash\overline{\mathtt{V}\ \text{ok}} \qquad \Delta;\Gamma\vdash\overline{\mathtt{e}:\mathtt{S}} \qquad \mathit{warning}}{\Delta;\Gamma\vdash\mathtt{e.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}:\star}\ (\text{T-Bot-Invk})$$

$$\frac{\Delta;\Gamma\vdash\mathtt{e}:\star \qquad \Delta;\Gamma\vdash\overline{\mathtt{e}:\mathtt{S}} \qquad \mathit{warning}}{\Delta;\Gamma\vdash\mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)}:\star}\ (\text{T-Bot-InvkR})$$

$$\frac{\Delta;\Gamma\vdash\mathtt{e}:\star \qquad \Delta\vdash\overline{\mathtt{V}\ \text{ok}} \qquad \Delta;\Gamma\vdash\overline{\mathtt{e}:\mathtt{S}} \qquad \mathit{warning}}{\Delta;\Gamma\vdash\mathtt{e.m[}\overline{\mathtt{a}}\mathtt{]<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}:\star}\ (\text{T-Bot-InvkA})$$

$$\frac{\Delta;\Gamma\vdash\mathtt{e}:\star \qquad \Delta;\Gamma\vdash\overline{\mathtt{e}:\mathtt{S}} \qquad \mathit{warning}}{\Delta;\Gamma\vdash\mathtt{e.m[}\overline{\mathtt{a}}\mathtt{]\,(}\overline{\mathtt{e}}\mathtt{)}:\star}\ (\text{T-Bot-InvkRA})$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash \text{ok} \qquad \Gamma(\mathtt{proceed})=\mathtt{R(}\overline{\mathtt{U}}\mathtt{)}\\ \Delta;\Gamma\vdash\overline{\mathtt{e}:\mathtt{S}} \qquad \Delta\vdash\overline{\mathtt{S}<:\mathtt{U}}\end{array}}{\Delta;\Gamma\vdash\mathtt{proceed(}\overline{\mathtt{e}}\mathtt{)}:\mathtt{R}}\ (\text{T-Proceed})$$

Figure 2: Expression typing

The substitution of `proceed` terms in expressions is different from substitution of other variables. This occurs because of the need to save the parameters of the method being advised that are not unified by the pointcut due to wildcard action. These extra parameters are concatenated with the actual parameters of `proceed` by means of the special substitution cases presented below. For the other cases, the substitution does nothing special.

$$[\mathtt{v.m[}\overline{\mathtt{a}}\mathtt{]\,(}\overline{\mathtt{d}}\mathtt{)/proceed}]\ \mathtt{proceed(}\overline{\mathtt{e}}\mathtt{)} = \ \mathtt{v.m[}\overline{\mathtt{a}}\mathtt{]\,(}\overline{\mathtt{e}}\mathtt{,}\overline{\mathtt{d}}\mathtt{)}$$

$$[\mathtt{v.m<}\overline{\mathtt{T}}\mathtt{>[}\overline{\mathtt{a}}\mathtt{]\,(}\overline{\mathtt{d}}\mathtt{)/proceed}]\ \mathtt{proceed(}\overline{\mathtt{e}}\mathtt{)} = \ \mathtt{v.m<}\overline{\mathtt{T}}\mathtt{>[}\overline{\mathtt{a}}\mathtt{]\,(}\overline{\mathtt{e}}\mathtt{,}\overline{\mathtt{d}}\mathtt{)}$$

When the advice list is empty, the rules R-Invk and R-InvkR provide normal method evaluation, substituting parameters for values and the `this` variable for the object where the method is being executed.

Type Erasure

$$\begin{array}{ll|ll} \mathit{bound}_\Delta(\texttt{X}) = \Delta(\texttt{X}) & & \mathit{head}(\texttt{C<T>}) = \texttt{C} \\ \mathit{bound}_\Delta(\texttt{N}) = \texttt{N} & & \mathit{head}(\texttt{C}) = \texttt{C} \end{array}$$

$$|\texttt{T}|_\Delta = \mathit{head}(\mathit{bound}_\Delta(\texttt{T}))$$

Field types fetching

$$\mathit{fields}(\texttt{Object}) = \varnothing$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{\}} \quad \mathit{fields}([\overline{\texttt{T/X}}]\texttt{N}) = \overline{\texttt{fn}}}{\mathit{fields}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{fn}}, [\overline{\texttt{T/X}}]\overline{\texttt{fd}}}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>} \triangleleft \texttt{N \{}\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{\}} \quad \mathit{fields}(\texttt{N}) = \overline{\texttt{fn}}}{\mathit{fields}(\texttt{C}) = |\overline{\texttt{fn}}, \overline{\texttt{fd}}|_{\overline{\texttt{X<:N}}}}$$

Constructor argument types fetching

$$\mathit{cargtype}(\texttt{Object}) = \varnothing$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>}\triangleleft\texttt{N\{}\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{\}} \quad \mathit{cargtype}([\overline{\texttt{T/X}}]\texttt{N}) = \overline{\texttt{fn}}}{\mathit{cargtype}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{fn}}, [\overline{\texttt{T/X}}]\overline{\texttt{fd}}}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>}\triangleleft\texttt{N\{}\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{\}} \quad \mathit{cargtype}(\texttt{N}) = \overline{\texttt{fn}} \\ \mathit{warning} \text{ if } |\overline{\texttt{fn}}, \overline{\texttt{fd}}|_\Delta \neq \overline{\texttt{fn}}, \overline{\texttt{fd}}\end{array}}{\mathit{cargtype}(\texttt{C}) = |\overline{\texttt{fn}}, \overline{\texttt{fd}}|_{\overline{\texttt{X<:N}}}}$$

Cooked method type fetching

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>} \triangleleft \texttt{N \{ }\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{ \}} \\ \texttt{<}\overline{\texttt{Y}\triangleleft\texttt{P}}\texttt{> R m (}\overline{\texttt{U x}}\texttt{) \{ e \}} \in \overline{\texttt{md}}\end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T/X}}]\texttt{<}\overline{\texttt{Y}\triangleleft\texttt{P}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{R}}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>} \triangleleft \texttt{N \{ }\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{md}}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \mathit{mtype}(\texttt{m}, [\overline{\texttt{T/X}}]\texttt{N})}$$

Raw method type fetching

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>} \triangleleft \texttt{N \{ }\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{ \}} \\ \texttt{<}\overline{\texttt{Y}\triangleleft\texttt{P}}\texttt{> R m (}\overline{\texttt{U x}}\texttt{) \{ e \}} \in \overline{\texttt{md}} \\ \Delta = \overline{\texttt{X<:N}}, \overline{\texttt{Y<:P}} \qquad \mathit{warning} \text{ if } |\overline{\texttt{U}}|_\Delta \neq \overline{\texttt{U}}\end{array}}{\mathit{mtyperaw}(\texttt{m}, \texttt{C}) = |\overline{\texttt{U}}|_\Delta \rightarrow |\texttt{R}|_\Delta}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>} \triangleleft \texttt{D<}\overline{\texttt{T}}\texttt{> \{ }\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{md}} \\ \mathit{mtype}(\texttt{m}, \texttt{D<}\overline{\texttt{T}}\texttt{>}) = \texttt{<}\overline{\texttt{Y}\triangleleft\texttt{P}}\texttt{> }\overline{\texttt{U}} \rightarrow \texttt{R} \\ \Delta = \overline{\texttt{X<:N}}, \overline{\texttt{Y<:P}} \qquad \mathit{warning} \text{ if } |\overline{\texttt{U}}|_\Delta \neq \overline{\texttt{U}}\end{array}}{\mathit{mtyperaw}(\texttt{m}, \texttt{C}) = |\overline{\texttt{U}}|_\Delta \rightarrow |\texttt{R}|_\Delta}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\triangleleft\texttt{N}}\texttt{>} \triangleleft \texttt{D \{ }\overline{\texttt{fd}}\texttt{ k }\overline{\texttt{md}}\texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{md}}}{\mathit{mtyperaw}(\texttt{m}, \texttt{C}) = \mathit{mtyperaw}(\texttt{m}, \texttt{D})}$$

Figure 3: Auxiliar definitions for type system rules

3 Properties of Raw-AFGJ

Since Raw-AFGJ was built on top of existing extensions for FGJ, here we point out some of its distinctive characteristics when compared to both Raw-FGJ and AFGJ.

matching: the pointcut matching is based on the logic introduced by Jagadeesan *et al.*[Jagadeesan et al., 2006]. This is a very simple monotonic logic that provides exact matching for pointcuts. We maintained this characteristic in Raw-AFGJ modifying only the way it is called in the *advisedby* relation (defined in Section 2.2.4) to represent the effect of matching against raw types.

conditional type erasure: The AspectJ weaver checks if a defined method signature does not contains type variables and then tries an exact match over this signature. If the method contains type variables, type erasure is

Evaluation context

$$\begin{array}{rl} \zeta ::= & [\bullet] \\ | & \zeta\mathtt{.f} \\ | & \zeta\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \\ | & \mathtt{v.m<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{v}}\mathtt{,}\zeta\mathtt{,}\overline{\mathtt{e}}\mathtt{)} \\ | & \zeta\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \\ | & \mathtt{v.m(}\overline{\mathtt{v}}\mathtt{,}\zeta\mathtt{,}\overline{\mathtt{e}}\mathtt{)} \\ | & \zeta\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>[}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{e}}\mathtt{)} \\ | & \mathtt{v.m<}\overline{\mathtt{T}}\mathtt{>[}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{v}}\mathtt{,}\zeta\mathtt{,}\overline{\mathtt{e}}\mathtt{)} \\ | & \zeta\mathtt{.m[}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{e}}\mathtt{)} \\ | & \mathtt{v.m[}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{v}}\mathtt{,}\zeta\mathtt{,}\overline{\mathtt{e}}\mathtt{)} \\ | & \mathtt{proceed(}\overline{\mathtt{v}}\mathtt{,}\zeta\mathtt{,}\overline{\mathtt{e}}\mathtt{)} \end{array}$$

Reduction rules

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_1 \longrightarrow \mathtt{e}_2}{\Delta;\Gamma \vdash \zeta\mathtt{[e}_1\mathtt{]} \longrightarrow \zeta\mathtt{[e}_2\mathtt{]}} \ (\text{R-EvalCtx})$$

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T\ f}} \qquad \mathtt{f}_i \in \overline{\mathtt{f}}}{\mathtt{new\ N(}\overline{\mathtt{v}}\mathtt{).f}_i \longrightarrow \mathtt{v}_i} \ (\text{R-Field})$$

$$\frac{\mathtt{[}\overline{\mathtt{a}}\mathtt{]} = [\mathtt{a} \mid \ \mathtt{a} \in \overline{\mathtt{ad}} \ \wedge \ \mathit{head}(\mathtt{N})\mathtt{.m}\ \mathit{advisedby}\ \mathtt{a}]}{\mathtt{new\ N(}\overline{\mathtt{v}}\mathtt{).m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{u}}\mathtt{)} \longrightarrow \mathtt{new\ N(}\overline{\mathtt{v}}\mathtt{).m<}\overline{\mathtt{V}}\mathtt{>[}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{u}}\mathtt{)}} \ (\text{R-Weave})$$

$$\frac{\mathtt{[}\overline{\mathtt{a}}\mathtt{]} = [\mathtt{a} \mid \ \mathtt{a} \in \overline{\mathtt{ad}} \ \wedge \ \mathit{head}(\mathtt{N})\mathtt{.m}\ \mathit{advisedby}\ \mathtt{a}]}{\mathtt{new\ N(}\overline{\mathtt{v}}\mathtt{).m(}\overline{\mathtt{u}}\mathtt{)} \longrightarrow \mathtt{new\ N(}\overline{\mathtt{v}}\mathtt{).m[}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{u}}\mathtt{)}} \ (\text{R-WeaveR})$$

$$\frac{\mathtt{advice\ a\ T\ (}\overline{\mathtt{T\ x}}\mathtt{)}:\phi\mathtt{\{\ e\ \}}}{\begin{array}{c}\mathtt{v.m<}\overline{\mathtt{V}}\mathtt{>[a,}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{u}}\mathtt{,}\overline{\mathtt{w}}\mathtt{)} \\ \longrightarrow \\ [\ \overline{\mathtt{u/x}},\ \mathtt{v/target},\ \mathtt{v.m<}\overline{\mathtt{V}}\mathtt{>[}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{w}}\mathtt{)/proceed}\ ]\mathtt{e}\end{array}} \ (\text{R-Advice})$$

$$\frac{\mathtt{advice\ a\ T\ (}\overline{\mathtt{T\ x}}\mathtt{)}:\phi\mathtt{\{\ e\ \}}}{\begin{array}{c}\mathtt{v.m[a,}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{u}}\mathtt{,}\overline{\mathtt{w}}\mathtt{)} \\ \longrightarrow \\ [\ \overline{\mathtt{u/x}},\ \mathtt{v/target},\ \mathtt{v.m[}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{w}}\mathtt{)/proceed}\ ]\mathtt{e}\end{array}} \ (\text{R-AdviceR})$$

$$\frac{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},\mathtt{N}) = (\overline{\mathtt{x}},\mathtt{e})}{\mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).m<}\overline{\mathtt{V}}\mathtt{>[](}\overline{\mathtt{d}}\mathtt{)} \longrightarrow [\ \overline{\mathtt{d/x}},\ \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)/this}\ ]\mathtt{e}} \ (\text{R-Invk})$$

$$\frac{\mathit{mbodyraw}(\mathtt{m},\mathtt{N}) = (\overline{\mathtt{x}},\mathtt{e})}{\mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).m[](}\overline{\mathtt{d}}\mathtt{)} \longrightarrow [\ \overline{\mathtt{d/x}},\ \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)/this}\ ]\mathtt{e}} \ (\text{R-InvkR})$$

Cooked method body fetching

$$\frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}\triangleleft\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{\ }\overline{\mathtt{fd}}\mathtt{\ k\ }\overline{\mathtt{md}}\mathtt{\ \}} \\ \mathtt{<}\overline{\mathtt{Y}\triangleleft\mathtt{P}}\mathtt{>\ R\ m\ (}\overline{\mathtt{U\ x}}\mathtt{)\ \{\ e\ \}} \in \overline{\mathtt{md}}\end{array}}{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = (\overline{\mathtt{x}},[\overline{\mathtt{T/X}},\overline{\mathtt{V/Y}}]\mathtt{e})}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}\triangleleft\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{\ }\overline{\mathtt{fd}}\mathtt{\ k\ }\overline{\mathtt{md}}\mathtt{\ \}} \qquad \mathtt{m} \notin \overline{\mathtt{md}}}{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},[\overline{\mathtt{T/X}}]\mathtt{N})}$$

Raw method body fetching

$$\frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}\triangleleft\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{\ }\overline{\mathtt{fd}}\mathtt{\ k\ }\overline{\mathtt{md}}\mathtt{\ \}} \\ \mathtt{<}\overline{\mathtt{Y}\triangleleft\mathtt{P}}\mathtt{>\ R\ m\ (}\overline{\mathtt{U\ x}}\mathtt{)\ \{\ e\ \}} \in \overline{\mathtt{md}}\end{array}}{\mathit{mbodyraw}(\mathtt{m},\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = (\overline{\mathtt{x}},[\overline{\mathtt{T/X}},\overline{\mathtt{*/Y}}]\mathtt{e})}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}\triangleleft\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{T}}\mathtt{>\ \{\ }\overline{\mathtt{fd}}\mathtt{\ k\ }\overline{\mathtt{md}}\mathtt{\ \}} \qquad \mathtt{m} \notin \overline{\mathtt{md}}}{\mathit{mbodyraw}(\mathtt{m},\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathit{mbodyraw}(\mathtt{m},[\overline{\mathtt{T/X}}]\mathtt{N})}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}\triangleleft\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{T}}\mathtt{>\ \{\ }\overline{\mathtt{fd}}\mathtt{\ k\ }\overline{\mathtt{md}}\mathtt{\ \}} \qquad \#(\overline{\mathtt{X}}) = \#(\overline{\mathtt{*}})}{\mathit{mbodyraw}(\mathtt{m},\mathtt{C}) = \mathit{mbodyraw}(\mathtt{m},\mathtt{C<}\overline{\mathtt{*}}\mathtt{>})}$$

Figure 4: Operational semantics

applied and the matching is done against its result[1]. In Raw-AFGJ, this behavior is modelled by the $\|\_\|_\Delta$, *matcheq* and *mtypeadv* operations, defined in Section 2.2.4.

concrete advices: AspectJ allows the definition of type parameters for abstract aspects. On the other hand, it does not allow the weaving of those aspects; only concrete aspects with all of their type parameters fully defined

[1] Note that the Java compiler saves generic information about *declared* attributes and method signatures as comments into the bytecode. Those comments are used by AspectJ to identify the declared generic type information needed during the weaving time.

are weaved. Since this feature does not interfere our goals (i.e. raw types and type erasure in aspect-oriented languages), we decided to remove parameterization from Raw-AFGJ advices in order to simplify the semantics (Section 2.1).

direct representation of raw types in pointcuts: In our calculus, both raw and cooked types can be freely used in pointcut expressions, with one notable exception: only raw types can be used as `target` classes in pointcuts. This limitation was originally introduced in AspectJ (due to the type erasure design) and it is directly represented in Raw-AFGJ through its abstract syntax.

Like in other related calculi, the type system of Raw-AFGJ is sound with respect to its semantics if *warnings* do not occur. Since we use structural operational semantics, type soundness comes from progress and preservation:

Theorem 1 (Progress): if $\varnothing;\varnothing \vdash \overline{\texttt{cd}}\ \overline{\texttt{ad}}\ \texttt{e} : \texttt{T}$ and $\overline{\texttt{ad}};\overline{\texttt{cd}} \vdash \text{ok}$ and $\neg$*warning*, then $\texttt{e}$ is a value or $\varnothing;\varnothing \vdash \overline{\texttt{cd}}\ \overline{\texttt{ad}}\ \texttt{e} \longrightarrow \overline{\texttt{cd}}\ \overline{\texttt{ad}}\ \texttt{e'}$.

Theorem 2 (Preservation): if $\varnothing;\varnothing \vdash \texttt{e} \longrightarrow \texttt{e'}$ and $\varnothing;\varnothing \vdash \overline{\texttt{cd}}\ \overline{\texttt{ad}}\ \texttt{e} : \texttt{T}$ and $\overline{\texttt{ad}};\overline{\texttt{cd}} \vdash \text{ok}$ and $\neg$*warning*, then $\varnothing;\varnothing \vdash \overline{\texttt{cd}}\ \overline{\texttt{ad}}\ \texttt{e'} : \texttt{S}$ and $\varnothing \vdash \texttt{S} <: \texttt{T}$.

Both proofs follow the standard syntactic soundness technique and are presented at the APPENDIX.

4 Related work

Featherweight Java (FJ), proposed by Igarashi, was developed to be a tool for studying extensions and variations of Java. This possibility was illustrated by two extensions: (1) Featherweight Generic Java (FGJ) [Igarashi et al., 2001a] which supports parametric polymorphism; and (2) Raw Featherweight Generic Java (Raw-FGJ) [Igarashi et al., 2001b] that extends FGJ with *raw types.*

Recently, some aspect-oriented calculi were proposed to study the influence of aspects over object-oriented languages [Clifton and Leavens, 2006, Jagadeesan et al., 2006], and also to study aspects in isolation [Bruns et al., 2004, Hui and Riely, 2007], i.e. independently of programming paradigm. In [Jagadeesan et al., 2006] a lightweight calculus is proposed in order to accommodate the subtleties of the interaction of classes, polymorphism and aspects. The resulting calculus, called Aspect Featherweight Generic Java (AFGJ), is an aspect-oriented extension of a cast-free version of FGJ. The pointcut language defined by the authors is very simple. It captures method `execution` and includes logic operators (except the `not` operator). They also explore two kinds of parametric polymorphism: the type erasure semantics of GJ, and the type carrying semantics of languages such as C#.

Even though AFGJ explores some subtleties of type erasure semantics, it does not mention *raw types*. Also, some restrictions were imposed over the pointcut structure disallowing non variable parameters for types in target objects. In our calculus, *type erasure* and *raw types* are treated both in the type system and in the semantics for both pointcut and Java definitions.

In [Fraine et al., 2008], an analysis of the type systems of current aspect oriented languages and how they can cause type errors is presented. They identify the implicit *proceed* signature as the main responsible for those undetected errors and they propose the *StrongAspectJ* language as a type safe extension of AspectJ with explicit *proceed* signature, in addition to regular advices. They included type ranges in pointcut signatures and type variables for generic advices. Like AFGJ, the *StrongAspectJ* type system does not consider *raw types*.

5 Conclusion

In this work we introduced Raw-AFGJ, an aspect-oriented calculus that reproduces the behavior of AspectJ with relation to raw types and type erasure. In particular, with Raw-AFGJ we can describe how AspectJ-like languages are influenced by generic Java design and how aspect weaving works with raw types and with the lack of type information at runtime.

We started by giving concrete examples illustrating how type erasure and raw types interact in subtle ways with aspect weaving semantics. We then proceeded by giving an operational semantics and a type system for Raw-AFGJ. This formal treatment was crucial for giving us a better understanding of the intricacies of AspectJ-like languages in the presence of raw types and type erasure. Besides proving the classical type soundness property for our calculus we also summarize its distinctive features when compared to other proposals for generic aspect oriented calculi.

As future work, we want to explore type rules for wildcards and inter-type declarations. We also are planning to use Raw-AFGJ to assure the correctness of some software refactorings related to generic types in AspectJ.

Acknowledgements

The authors would like to thank CAPES for the given support, and the anonymous referees for their comments and suggestions.

References

[Bracha et al., 1998] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998). Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA. ACM Press.

[Bruns et al., 2004] Bruns, G., Jagadeesan, R., Jeffrey, A., and Riely, J. (2004). μabc: A minimal aspect calculus. In Gardner, P. and Yoshida, N., editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer.

[Clifton and Leavens, 2006] Clifton, C. and Leavens, G. T. (2006). Minimao1: an imperative core language for studying aspect-oriented reasonings. *Sci. Comput. Program.*, 63(3):321–374.

[Fraine et al., 2008] Fraine, B. D., Südholt, M., and Jonckers, V. (2008). StrongAspectJ: flexible and safe pointcut/advice bindings. In D'Hondt, T., editor, *AOSD*, pages 60–71. ACM.

[Hui and Riely, 2007] Hui, P. and Riely, J. (2007). Typing for a minimal aspect language: preliminary report. In *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*, pages 15–22, New York, NY, USA. ACM.

[Igarashi et al., 2001a] Igarashi, A., Pierce, B. C., and Wadler, P. (2001a). Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450.

[Igarashi et al., 2001b] Igarashi, A., Pierce, B. C., and Wadler, P. (2001b). A recipe for raw types. In *Workshop on Foundations of Object-Oriented Languages (FOOL).*

[Jagadeesan et al., 2006] Jagadeesan, R., Jeffrey, A., and Riely, J. (2006). Typed parametric polymorphism for aspects. *Sci. Comput. Program.*, 63(3):267–296.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer.

Appendix – Proofs

Theorem 1 (Progress) if $\emptyset;\emptyset \vdash \overline{\texttt{cd}}\ \overline{\texttt{ad}}\ \texttt{e}:\texttt{T}$ and $\overline{\texttt{ad}};\overline{\texttt{cd}} \vdash \text{ok}$ and $\neg$ *warning* then $\texttt{e}$ is a value or $\exists\texttt{e'}.\emptyset;\emptyset \vdash \overline{\texttt{cd}}\ \overline{\texttt{ad}}\ \texttt{e} \longrightarrow \overline{\texttt{cd}}\ \overline{\texttt{ad}}\ \texttt{e'}$.

Proof: By induction on the structure of type derivations for expressions.

Cases whose type is $\star$, typed by T-Bot-Field, T-Bot-Invk, T-Bot-InvkR,T-Bot-InvkA, T-Bot-InvkRA.

- these rules can never occur, since they depend on *warning*, which can not occur by the premise of the progress.

Case $\texttt{e} \equiv \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$**,** where $\texttt{e}$ is typed by T-New.

- Are all parameters $\overline{\texttt{e}}$ values?
  - Yes. Then $\texttt{e}$ is a value.
  - No. Then, at least one of $\overline{\texttt{e}}$ is not a value. Take $\texttt{e}_i$ to be the leftmost non-value expression of $\overline{\texttt{e}}$. Taking ζ to be $\texttt{v.m<}\overline{\texttt{T}}\texttt{>[}\overline{\texttt{a}}\texttt{](}\overline{\texttt{v}},\zeta,\overline{\texttt{e}}\texttt{)}$, then $\texttt{e} \equiv \zeta[\texttt{e}_i]$. Using the inductive hypothesis, there is a $\texttt{e}'_i$ such that $\Delta;\Gamma \vdash \texttt{e}_i \longrightarrow \texttt{e}'_i$. Therefore, progress is assured by R-EvalCtx.
    Since this argument appears quite often, we will simply point it in later cases by "Progress by R-EvalCtx using the inductive hypothesis".

Case $\texttt{e} \equiv \texttt{new C(}\overline{\texttt{e}}\texttt{)}$, $\texttt{e}$ is typed by T-NewR.

- Are all parameters $\overline{\texttt{e}}$ values?
  - Yes. Then $\texttt{e}$ is a value.
  - No. Progress by R-EvalCtx using the inductive hypothesis.

Case $\texttt{e} \equiv \texttt{x}$, $\texttt{e}$ is typed by T-Var.

- The first premise of progress states that $\Gamma = \varnothing$ and all typing rules for expressions keep Γ unchanged. Therefore, this case will never occur in the derivation, since $\Gamma(x) = \texttt{T}$ is needed by T-Var.

Case $\texttt{e} \equiv \texttt{d.f}$, `e` is typed by T-Field.

- Is `d` a value?
  - Yes. Then, by the premises of T-Field, $\texttt{f} \in \mathit{fields}(\texttt{d})$. Therefore, `d.f` can be reduced by R-Field.
  - No. Progress by R-EvalCtx using the inductive hypothesis.

Case $\texttt{e} \equiv \texttt{d.m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)}$, `e` is typed by T-Invk.

- Is `d` a value?
  - Yes. Are all elements of $\overline{\texttt{d}}$ values?
    - Yes. Then, the expression is reduced by R-Weave (it is always possible to build a list of all advices that intercept the given method invocation).
    - No. Progress by R-EvalCtx using the inductive hypothesis.
  - No. Progress by R-EvalCtx using the inductive hypothesis.

Case $\texttt{e} \equiv \texttt{e.m(}\overline{\texttt{d}}\texttt{)}$, `e` is typed by T-InvkR.

- Is `d` a value?
  - Yes. Are all elements of $\overline{\texttt{d}}$ values?
    - Yes. Then, the expression is reduced by R-WeaveR (it is always possible to build a list of all advices that intercept the given method invocation).
    - No. Progress by R-EvalCtx using the inductive hypothesis.
  - No. Progress by R-EvalCtx using the inductive hypothesis.

Case $\texttt{e} \equiv \texttt{d.m<}\overline{\texttt{V}}\texttt{>[}\overline{\texttt{a}}\texttt{](}\overline{\texttt{d}}\texttt{)}$, `e` is typed by T-InvkA.

- Is `d` a value?
  - Are all elements of `d` values?
    - Yes. Is the list $\overline{\texttt{a}}$ empty?
      - Yes. Since `d` is a value, it is of non-variable type `N` by `T-New` or `T-NewR`. By T-InvkA, $\mathit{mtype}(\texttt{m},\texttt{N})$ is defined. By Lemma 3, $\mathit{dom}(\mathit{mtype}) = \mathit{dom}(\mathit{mbody})$, therefore $\mathit{mbody}(\texttt{m},\texttt{N})$ is also defined. This assures progress by rule R-Invk.
      - No. By T-InvkA, all advice names in $\overline{\texttt{a}}$ intercept the method call by means of the *advisedby* relation. The *advisedby* relation only holds for declared advice names. Therefore, the expression is reduced by means of R-Advice.
    - No. Progress by R-EvalCtx using the inductive hypothesis.
  - No. Progress by R-EvalCtx using the inductive hypothesis.

Case $\texttt{e} \equiv \texttt{d.m[}\overline{\texttt{a}}\texttt{](}\overline{\texttt{d}}\texttt{)}$, `e` is typed by T-InvkRA.

- Is `d` a value?
  - Are all elements of `d` values?
    - Yes. Is the list $\overline{\texttt{a}}$ empty?
      - Yes. Since `d` is a value, it is of non-variable type `N` by `T-New` or `T-NewR`. By T-InvkRA, $\mathit{mtyperaw}(\texttt{m}, |\texttt{N}|_\Delta)$ is defined. By Lemma 4, $\mathit{dom}(\mathit{mtyperaw}) = \mathit{dom}(\mathit{mbodyraw})$, therefore $\mathit{mbodyraw}(\texttt{m}, |\texttt{N}|_\Delta)$ is also defined. This assures progress by rule R-InvkR.
      - No. By T-InvkRA, all advice names in $\overline{\texttt{a}}$ intercept the method call by means of the *advisedby* relation. The *advisedby* relation only holds for declared advice names. Therefore, the expression is reduced by means of R-AdviceR.
    - No. Progress by R-EvalCtx using the inductive hypothesis.
  - No. Progress by R-EvalCtx using the inductive hypothesis.

Case $\texttt{e} \equiv \texttt{proceed(}\overline{\texttt{e}}\texttt{)}$, `e` is typed by T-Proceed.

– The first premise of progress states that $\Gamma = \varnothing$ and all typing rules for expressions keep Γ unchanged. Therefore, this case will never occur in the derivation, since $\Gamma(\texttt{proceed}) = \texttt{R}(\overline{\texttt{U}})$ is needed by T-PROCEED.

Theorem 2 (Preservation) if $\varnothing;\varnothing \vdash \texttt{e} \longrightarrow \texttt{e'}$ and $\varnothing;\varnothing \vdash \texttt{e} : \texttt{T}$ and $\overline{\texttt{ad}};\overline{\texttt{cd}} \vdash \text{ok}$ and $\neg$ *warning*, then $\varnothing;\varnothing \vdash \texttt{e'} : \texttt{S}$ and $\varnothing \vdash \texttt{S} <: \texttt{T}$.

Proof: by induction on the structure of the reduction relation.

Case $\texttt{e} \longrightarrow \texttt{e'}$ by R-EVALCTX, $\texttt{e} \equiv \zeta\texttt{[d]}$, $\texttt{e}' \equiv \zeta\texttt{[d']}$.
By the premise of R-EVALCTX, $\varnothing;\varnothing \vdash \texttt{d} \longrightarrow \texttt{d'}$. All typing rules rely on the typing of its subexpressions to provide the type of a given expression. Therefore, if $\zeta\texttt{[d]}$ is well-typed, then the fact that $\texttt{d}$ is well-typed is one of its premises. Hence, $\zeta\texttt{[d']}$ can be typed using the same type rule as $\zeta\texttt{[d]}$.

Case $\texttt{e} \longrightarrow \texttt{e'}$ by R-FIELD, $\texttt{e} \equiv \texttt{new N(}\overline{\texttt{v}}\texttt{).f}_i$, $\texttt{e}' \equiv \texttt{v}_i$.
In this case, $\texttt{e}$ is a value typed by T-NEW or T-NEWR. In both type rules, the types for the elements of the value list $\overline{\texttt{v}}$ are insecure subtypes of the result of *cargtype*(N). The function *cargtype* has the same behavior as the *fields* function but for signaling a *warning* if some type of the field list is modified by the raw class type fetching. By the premises of preservation we assure $\neg$*warning*, which causes *fields*(N) = *cargtype*(N). Therefore, we have that $\Delta;\Gamma \vdash \texttt{v}_i : \texttt{S}_i$ and $\Delta \vdash \texttt{S}_i <:_? \texttt{T}_i$. Since $\neg$*warning*, then also $\Delta \vdash \texttt{S}_i <: \texttt{T}_i$.

Case $\texttt{e} \longrightarrow \texttt{e'}$ by R-WEAVE, $\texttt{e} \equiv \texttt{new N(}\overline{\texttt{v}}\texttt{).m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{u}}\texttt{)}$, $\texttt{e}' \equiv \texttt{new N(}\overline{\texttt{v}}\texttt{).m<}\overline{\texttt{V}}\texttt{>[}\overline{\texttt{a}}\texttt{](}\overline{\texttt{u}}\texttt{)}$.
In this case, $\texttt{e}$ is typed by T-INVK. All elements of the list $\overline{\texttt{a}}$ of advice names are related to the given method call by means of *advisedby*. The premises of T-INVK and R-WEAVE allows the result $\texttt{e'}$ to be typed by means of T-INVKA with the same type as $\texttt{e}$ (the only difference between T-INVK and T-INVKA is the requirement of matching for the incident advices).

Case $\texttt{e} \longrightarrow \texttt{e'}$ by R-WEAVER.
Similar to the previous case, considering T-INVKR, R-WEAVER and T-INVKRA.

Case $\texttt{e} \longrightarrow \texttt{e'}$ by R-ADVICE, $\texttt{e} \equiv \texttt{v.m<}\overline{\texttt{V}}\texttt{>[a,}\overline{\texttt{a}}\texttt{](}\overline{\texttt{u}}\texttt{,}\overline{\texttt{w}}\texttt{)}$, $\texttt{e'} \equiv [\,\overline{\texttt{u/x}}, \texttt{v/target}, \texttt{v.m<}\overline{\texttt{V}}\texttt{>[}\overline{\texttt{a}}\texttt{](}\overline{\texttt{w}}\texttt{)/proceed}\,]\texttt{d}$.
The term $\texttt{e}$ is typed by T-INVKA. By the premises of T-INVKA, the parameters $\overline{\texttt{u}}$ are of a subtype of the corresponding result of *mtype*. By the premises of T-INVKA the advice with name $\texttt{a}$ is defined in $\overline{\texttt{ad}}$. By $\overline{\texttt{ad}};\overline{\texttt{cd}} \vdash \text{ok}$, all defined advices are well-formed by T-ADV. By the premises of T-ADV, $\varnothing;\overline{\texttt{T x}},\texttt{C target},\texttt{T proceed(}\overline{\texttt{T}}\texttt{)} \vdash \texttt{d} : \texttt{T'}$ and $\varnothing \vdash \texttt{T'} <: \texttt{T}$, where $\texttt{T}$ is the return type of the advised method and $\overline{\texttt{T}}$ are the types of its parameters (due the exact match of the pointcut logic). Therefore, using Lemma 1 (preservation of substitution), we have $\varnothing;\varnothing \vdash [\,\overline{\texttt{u/x}}, \texttt{v/target}, \texttt{v.m<}\overline{\texttt{V}}\texttt{>[}\overline{\texttt{a}}\texttt{](}\overline{\texttt{w}}\texttt{)/proceed}\,]\texttt{d} : \texttt{T'}$ and $\varnothing \vdash \texttt{T'} <: \texttt{T}$.

Case $\texttt{e} \longrightarrow \texttt{e'}$ by R-ADVICER.
Similar to the previous case, considering T-INVKRA.

Case $\texttt{e} \longrightarrow \texttt{e'}$ by R-INVK, $\texttt{e} \equiv \texttt{new N(}\overline{\texttt{w}}\texttt{).m<}\overline{\texttt{V}}\texttt{>[](}\overline{\texttt{u}}\texttt{)}$, $\texttt{e'} \equiv [\,\overline{\texttt{u/x}}, \texttt{new N(}\overline{\texttt{w}}\texttt{)/this}\,]\texttt{d}$.
The term $\texttt{e}$ is typed by T-INVKA. The type of $\texttt{e}$ is $\texttt{[}\overline{\texttt{V/Y}}\texttt{]R}$, where *mtype*(m,N) = $\texttt{<}\overline{\texttt{Y}\triangleleft\texttt{P}}\texttt{>U}\rightarrow\texttt{R}$. We have two possibilities for N:

– Case $\texttt{N} \equiv \texttt{C<}\overline{\texttt{U}}\texttt{>}$.
By the definition of *mtype*, $\texttt{R} \equiv \texttt{[}\overline{\texttt{U/X}}\texttt{]R'}$, where $\overline{\texttt{X}}$ is the type parameter list for C. Therefore, the type of $\texttt{e}$ is $\texttt{[}\overline{\texttt{V/Y}}\texttt{]([}\overline{\texttt{U/X}}\texttt{]R')}$.
By the premises of R-INVK, *mbody*($\texttt{m<}\overline{\texttt{V}}\texttt{>,C<}\overline{\texttt{U}}\texttt{>}$) = ($\overline{\texttt{x}}$,d). By construction *mtype*($\texttt{m,C<}\overline{\texttt{U}}\texttt{>}$) and *mbody*($\texttt{m<}\overline{\texttt{V}}\texttt{>,C<}\overline{\texttt{U}}\texttt{>}$) will obtain its results from the same method definition $\texttt{md}$. By the good formation of the environments, T-METHOD holds for all declared methods including $\texttt{md}$. By the premises of T-METHOD, then $\varnothing,\overline{\texttt{X}\triangleleft\texttt{N}},\overline{\texttt{Y}\triangleleft\texttt{P}};\varnothing,\texttt{N this},\overline{\texttt{U u}} \vdash \texttt{d} : \texttt{S}$

and $\varnothing,\overline{\mathtt{X}\triangleleft\mathtt{N}},\overline{\mathtt{Y}\triangleleft\mathtt{P}} \vdash \mathtt{S} <: \mathtt{R'}$. By Lemma 2 (preservation of type variable substitution), then $\varnothing;\varnothing,[\overline{\mathtt{U/X}},\overline{\mathtt{V/Y}}]\mathtt{N}\ \mathtt{this},\overline{[\overline{\mathtt{U/X}},\overline{\mathtt{V/Y}}]\mathtt{U}\ \mathtt{u}} \vdash [\overline{\mathtt{U/X}},\overline{\mathtt{V/Y}}]\mathtt{d} : [\overline{\mathtt{U/X}},\overline{\mathtt{V/Y}}]\mathtt{S}$ holds, and also $\varnothing \vdash \overline{[\overline{\mathtt{U/X}},\overline{\mathtt{V/Y}}]\mathtt{S}} <: \overline{[\overline{\mathtt{U/X}},\overline{\mathtt{V/Y}}]\mathtt{R'}}$. By Lemma 1 (preservation of substitution), then $\varnothing;\varnothing \vdash$ $[\ \overline{[\overline{\mathtt{U/X}},\overline{\mathtt{V/Y}}]\mathtt{u/x}},\ \mathtt{new}\ [\overline{\mathtt{U/X}},\overline{\mathtt{V/Y}}]\mathtt{N}(\overline{\mathtt{w}})\mathtt{/this}\ ]\ [\overline{\mathtt{U/X}},\overline{\mathtt{V/Y}}]\mathtt{d} : [\overline{\mathtt{U/X}},\overline{\mathtt{V/Y}}]\mathtt{S}$.

– Case $\mathtt{N} \equiv \mathtt{C}$
  Construction of the proof is similar to the above case, but using *mtyperaw* and *mbodyraw*.

Case $\mathtt{e} \longrightarrow \mathtt{e'}$ by R-INVKR. $\mathtt{e} \equiv \mathtt{v.m[](}\overline{\mathtt{u}},\overline{\mathtt{w}}\mathtt{)}$, $\mathtt{e'} \equiv [\ \overline{\mathtt{d/x}},\ \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)/this}\ ]\mathtt{d}$.
  Similar to the previous case, considering T-INVKRA as the type rule for e. For the $|\ |_{\Delta}$ operation, Lemma 2 (preservation of type variable substitution) is used.

Lemma 1 (preservation of substitution)

Substitution of variables

The substitution of variable `x` for value `v` in expression `e`, denoted as `[v/x]e`, is defined as follows:

$\mathtt{[v/x]x}$	$= \mathtt{v}$
$\mathtt{[v/x]y}$	$= \mathtt{y}$ if $\mathtt{x} \neq \mathtt{y}$
$\mathtt{[v/x]new\ N(}\overline{\mathtt{e}}\mathtt{)}$	$= \mathtt{new\ N([v/x]}\overline{\mathtt{e}}\mathtt{)}$
$\mathtt{[v/x](e.f)}$	$= \mathtt{([v/x]e).f}$
$\mathtt{[v/x](e.m<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{))}$	$= \mathtt{([v/x]e).m<}\overline{\mathtt{T}}\mathtt{>([v/x]}\overline{\mathtt{e}}\mathtt{)}$
$\mathtt{[v/x](e.m(}\overline{\mathtt{e}}\mathtt{))}$	$= \mathtt{([v/x]e).m([v/x]}\overline{\mathtt{e}}\mathtt{)}$
$\mathtt{[v/x](e.m[}\overline{\mathtt{a}}\mathtt{]<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{))}$	$= \mathtt{([v/x]e).m[}\overline{\mathtt{a}}\mathtt{]<}\overline{\mathtt{T}}\mathtt{>([v/x]}\overline{\mathtt{e}}\mathtt{)}$
$\mathtt{[v/x](e.m[}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{e}}\mathtt{))}$	$= \mathtt{([v/x]e).m[}\overline{\mathtt{a}}\mathtt{]([v/x]}\overline{\mathtt{e}}\mathtt{)}$
$\mathtt{[v/x]proceed(}\overline{\mathtt{e}}\mathtt{)}$	$= \mathtt{proceed([x/v]}\overline{\mathtt{e}}\mathtt{)}$
$\mathtt{[v/x]}\overline{\mathtt{e}}$	$= \mathtt{[v/x]e}_1, \mathtt{[v/x]e}_2, \ldots, \mathtt{[v/x]e}_n$

The type preservation for substitution of variables is stated as:

if $\Delta;\Gamma,\mathtt{U\ x} \vdash \mathtt{e} : \mathtt{T}$ **and** $\Delta;\Gamma \vdash \mathtt{v} : \mathtt{U}'$ **and** $\Delta \vdash \mathtt{U'} <: \mathtt{U}$ **and** $\Delta;\Gamma \vdash \mathrm{ok}$, **then** $\Delta;\Gamma \vdash \mathtt{[v/x]e} : \mathtt{T'}$ **and** $\Delta \vdash \mathtt{T'} <: \mathtt{T}$.

Proof: by induction on the structure of the substitution.

Case $\mathtt{e} \equiv \mathtt{x}$.
  The expression `x` is typed by T-VAR, having type $\Gamma(\mathtt{x}) = \mathtt{U}$. By the premisses of preservation of substitution, $\Delta;\Gamma \vdash \mathtt{v} : \mathtt{U}'$ and $\Delta \vdash \mathtt{U'} <: \mathtt{U}$.

Other cases . The other cases simply propagate the substitution to its subexpressions. The preservation of types is assured by the inductive hypothesis over the substitution on subexpressions.

Substitution of proceed (cooked)

The substitution of the expression $\mathtt{proceed(}\overline{\mathtt{u}}\mathtt{)}$ for a cooked method call $\mathtt{v.m[}\overline{\mathtt{a}}\mathtt{]<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{w}}\mathtt{)}$ in expression `e`, denoted as $\mathtt{[v.m[}\overline{\mathtt{a}}\mathtt{]<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{w}}\mathtt{)/proceed]e}$, is defined as follows.

$$\begin{array}{ll}
[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{x} & = \texttt{x} \\
[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{new N(}\overline{\texttt{e}}\texttt{)} & = \texttt{new N(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\overline{\texttt{e}}\texttt{)} \\
[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{(e.f)} & = \texttt{(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{e).f} \\
[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{(e.m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{))} & = \texttt{(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{e).} \\
 & \qquad \texttt{m<}\overline{\texttt{T}}\texttt{>(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\overline{\texttt{e}}\texttt{)} \\
[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{(e.m(}\overline{\texttt{e}}\texttt{))} & = \texttt{(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{e).} \\
 & \qquad \texttt{m(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\overline{\texttt{e}}\texttt{)} \\
[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{(e.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{))} & = \texttt{(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{e).} \\
 & \qquad \texttt{m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\overline{\texttt{e}}\texttt{)} \\
[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{(e.m}[\overline{\texttt{a}}]\texttt{(}\overline{\texttt{e}}\texttt{))} & = \texttt{(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{e).} \\
 & \qquad \texttt{m}[\overline{\texttt{a}}]\texttt{(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\overline{\texttt{e}}\texttt{)} \\
[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{proceed(}\overline{\texttt{e}}\texttt{)} & = \texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>} \\
 & \qquad \texttt{(}[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\overline{\texttt{e}}\texttt{,}\overline{\texttt{w}}\texttt{)} \\
[\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\overline{\texttt{e}} & = [\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{e}_1, \\
 & \ [\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{e}_2, \\
 & \ \ldots, \\
 & \ [\texttt{v.m}[\overline{\texttt{a}}]\texttt{<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{e}_n
\end{array}$$

The type preservation for substitution of proceed (cooked) is stated as:

if $\Delta;\Gamma,\texttt{U proceed(}\overline{\texttt{V}}\texttt{)} \vdash \texttt{e} : \texttt{T}$ **and** $\Delta;\Gamma \vdash \texttt{v.m}[\overline{\texttt{a}}]\texttt{(}\overline{\texttt{v}}\texttt{,}\overline{\texttt{w}}\texttt{)} : \texttt{U}'$ **and** $\Delta;\Gamma,\texttt{U proceed(}\overline{\texttt{V}}\texttt{)} \vdash \overline{\texttt{v}:\texttt{V'}}$ **and** $\Delta \vdash \overline{\texttt{V'} <: \texttt{V}}$ **and** $\Delta \vdash \texttt{U'} <: \texttt{U}$ **and** $\Delta;\Gamma \vdash$ ok, **then** $\Delta;\Gamma \vdash [\texttt{v.m}[\overline{\texttt{a}}]\texttt{(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{e} : \texttt{T'}$ **and** $\Delta \vdash \texttt{T'} <: \texttt{T}$.

Proof: by induction on the structure of the substitution.

Case $\texttt{e} \equiv \texttt{proceed(}\overline{\texttt{d}}\texttt{)}$.
The expression $\texttt{proceed(}\overline{\texttt{d}}\texttt{)}$ is typed by T-Proceed. By the premises of T-Proceed, $\Gamma(\texttt{proceed}) = \texttt{U}(\overline{\texttt{V}})$, $\Delta;\Gamma,\texttt{U proceed(}\overline{\texttt{V}}\texttt{)} \vdash \overline{\texttt{d}:\texttt{V'}}$ and $\Delta \vdash \texttt{V'} <: \texttt{V}$. By the inductive hypothesis, $\Delta;\Gamma \vdash \texttt{d} : \texttt{V''}$ and $\Delta \vdash \texttt{V''} <: \texttt{V''}$. By transitivity of the subtype relation, $\Delta \vdash \texttt{V''} <: \texttt{V}$. This suffices to match the types of the first arguments of $\texttt{v.m}[\overline{\texttt{a}}]\texttt{(}\overline{\texttt{v}}\texttt{,}\overline{\texttt{w}}\texttt{)}$, which can be typed by `T-InvkA`.

Other cases . The other cases simply propagate the substitution to its subexpressions. The preservation of types is assured by the inductive hypothesis over the substitution on subexpressions.

Substitution of proceed (raw)

The substitution of the expression $\texttt{proceed(}\overline{\texttt{u}}\texttt{)}$ for a raw method call $\texttt{v.m}[\overline{\texttt{a}}]\texttt{(}\overline{\texttt{w}}\texttt{)}$ in `e`, denoted as $[\texttt{v.m}[\overline{\texttt{a}}]\texttt{(}\overline{\texttt{w}}\texttt{)/proceed}]\texttt{e}$, is stated in the same way as the cooked method counterpart.

Lemma 2 (preservation of type variable substitution)

The substitution of type variable `X` for type `V` in type `T`, denoted as `[V/X]T`, is defined as follows:

$$\begin{array}{ll}
\texttt{[V/X]X} & = \texttt{V} \\
\texttt{[V/X]Y} & = \texttt{Y} \quad \text{if } \texttt{X} \neq \texttt{Y} \\
\texttt{[V/X]C} & = \texttt{C} \\
\texttt{[V/X]C<}\overline{\texttt{T}}\texttt{>} & = \texttt{C<[V/X]}\overline{\texttt{T}}\texttt{>} \\
\texttt{[V/X]}\star & = \star \\
\texttt{[V/X]}\overline{\texttt{T}} & = \texttt{[V/X]T}_1, \texttt{[V/X]T}_2, \ldots, \texttt{[V/X]T}_n
\end{array}$$

The substitution of type variable `X` for type `V` in *expression* `e`, denoted as `[V/X]e`, is defined as follows:

$$\begin{aligned}
&\texttt{[V/X]x} &&= \texttt{x}\\
&\texttt{[V/X]new N(}\overline{\texttt{e}}\texttt{)} &&= \texttt{new [V/X]N([V/X]}\overline{\texttt{e}}\texttt{)}\\
&\texttt{[V/X](e.f)} &&= \texttt{([V/X]e).f}\\
&\texttt{[V/X](e.m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{))} &&= \texttt{([V/X]e).m<[V/X]}\overline{\texttt{T}}\texttt{>([V/X]}\overline{\texttt{e}}\texttt{)}\\
&\texttt{[V/X](e.m(}\overline{\texttt{e}}\texttt{))} &&= \texttt{([V/X]e).m([V/X]}\overline{\texttt{e}}\texttt{)}\\
&\texttt{[V/X](e.m[}\overline{\texttt{a}}\texttt{]<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{))} &&= \texttt{([V/X]e).m[}\overline{\texttt{a}}\texttt{]<[V/X]}\overline{\texttt{T}}\texttt{>([V/X]}\overline{\texttt{e}}\texttt{)}\\
&\texttt{[V/X](e.m[}\overline{\texttt{a}}\texttt{](}\overline{\texttt{e}}\texttt{))} &&= \texttt{([V/X]e).m[}\overline{\texttt{a}}\texttt{]([V/X]}\overline{\texttt{e}}\texttt{)}\\
&\texttt{[V/X]proceed(}\overline{\texttt{e}}\texttt{)} &&= \texttt{proceed([V/X]}\overline{\texttt{e}}\texttt{)}\\
&\texttt{[V/X]}\overline{\texttt{e}} &&= \texttt{[V/X]e}_1, \texttt{[V/X]e}_2, \ldots, \texttt{[v/x]e}_n
\end{aligned}$$

The substitution of type variable `X` for type `V` in *type environment* Γ, denoted as `[V/X]`Γ, is defined as follows:

$$\begin{aligned}
&\texttt{[V/X]}\varnothing &&= \varnothing\\
&\texttt{[V/X](}\Gamma\texttt{,T x)} &&= \texttt{([V/X]}\Gamma\texttt{),[V/X]T x}\\
&\texttt{[V/X](}\Gamma\texttt{,U proceed(}\overline{\texttt{T}}\texttt{))} &&= \texttt{([V/X]}\Gamma\texttt{),[V/X]U proceed([V/X]}\overline{\texttt{T}}\texttt{)}
\end{aligned}$$

The substitution of type variable `X` for type `V` in *type variable environment* Γ, denoted as `[V/X]`Δ, is defined as follows:

$$\begin{aligned}
&\texttt{[V/X]}\varnothing &&= \varnothing\\
&\texttt{[V/X](}\Delta\texttt{,X}\lhd\texttt{N)} &&= \texttt{([V/X]}\Delta\texttt{),X}\lhd\texttt{[V/X]N}
\end{aligned}$$

The preservation of types in type judgements after type variable substitution is stated as:

if $\Delta, \Gamma \vdash \text{ok}$ **and** $\Delta, \texttt{X}\lhd\texttt{N}; \Gamma \vdash \texttt{e} : \texttt{T}$ **and** $\texttt{[V/X]}\Delta \vdash \texttt{V} <: \texttt{N}$ **then** $\texttt{[V/X]}\Delta, \texttt{[V/X]}\Gamma \vdash \texttt{[V/X]e} : \texttt{[V/X]T'}$ **and** $\texttt{[V/X]}\Delta \vdash \texttt{[V/X]T'} <: \texttt{[V/X]T}$

Proof: by induction on the structure of the substitution over type judgements and the subtype relation.

Lemma 3 (mtype and mbody have the same domain)

if $\overline{\texttt{ad}}, \overline{\texttt{cd}} \vdash \text{ok}$ **then** $\textit{mtype}(\texttt{m,C<}\overline{\texttt{T}}\texttt{>})\ \textit{defined} \Leftrightarrow \textit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>,C<}\overline{\texttt{T}}\texttt{>})\ \textit{defined}$

Proof:

- $\texttt{C} \in \overline{\texttt{cd}}$?
  - No. Then both functions are undefined.
  - Yes. Let `class C<`$\overline{\texttt{X}\lhd\texttt{N}}$`>` $\lhd$`P {` $\overline{\texttt{ad}}$ `k` $\overline{\texttt{md}}$`}` be the definition of `C` in $\overline{\texttt{cd}}$. $\texttt{m} \in \overline{\texttt{md}}$?
    - Yes. Then both functions are defined.
    - No. Then the result of the functions *mtype* and *mbody* for the method `m` in `C` is the same as the result of the method `m` in `P` (after type variable substitution).

Since $\overline{\texttt{ad}}, \overline{\texttt{cd}} \vdash \text{ok}$, the class hierarchy is acyclic and has a top element. Therefore, the chain of recursive calls of both functions always finishes. If `m` is defined in some superclass of `C`, then both functions are defined. Otherwise, they are both undefined.

Lemma 4 (mtyperaw and mbodyraw have the same domain)

Similar statement and proof as Lemma 3.