On the Interaction of Advices and Raw Types in AspectJ

Fernando Barden Rubbo\(^1\) Rodrigo Machado\(^1\) Álvaro Freitas Moreira\(^1\) Leila Ribeiro\(^1\) Daltro José Nunes\(^1\)

\(^1\)Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
{fbrubbo, rma, afmoreira, leila, 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.

1. Introduction

The AspectJ language\(^9\) 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.

Since the Java Virtual Machine has not changed 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 cannot be said about their role in polymorphic AspectJ. As an example, consider the following piece of code:

```java
public class Buffer<E extends Object> {
    E content;
    void setContent(E n){content = n;}
    E getContent(){return content;}
    void foo(Buffer<Integer> a){}

    public static void main(String[] args) {
        Buffer<Integer> a = new Buffer<Integer>();
        Integer b;
        a.setContent(42);
        b = a.getContent();
        a.foo(a);
    }
}
```

```
public aspect TestRaw {
    before(): execution(Integer *(..)){
        System.out.println("This method \n returns an Integer\n");
    }
    before(): execution(* *(..))
        && args(Buffer<Integer>) {
        System.out.println("This method's argument \n has type Buffer<Integer>\n");
    }
}
```

The class `Buffer` represents a one-position container, whose content type is defined by means of type parameter `E`. Within the class there is one field `content` of type `E`, a setter method `setContent`, a getter method `getContent`, and a useless method `foo`. The class contains a `main` method, which simply instantiates `Buffer` with type parameter `Integer`, and execute calls to `setContent`, `getContent` and `foo`. Over this small program, there is an aspect that defines two advices: one that prints a message before the execution of any method that returns an `Integer`, and one that prints a message before the execution of any method that has exactly one parameter of type `Buffer<Integer>`. At first glance, it would be natural to think that both advices are triggered when the `main` method executes: one intercepting the execution of `getContents` (since it returns an `Integer`), and the other the execution of `foo`. However, only the second advice is executed in the actual AspectJ implementation. Due to type erasure, the type variable `E` is substituted by its bound (`Object`) during compilation. Consequently, the return type of `getContents` is actually `Object` in the bytecode. Since AspectJ provides bytecode-level weaving, the first advice fails to match with `getContents`, and it is not executed.

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

```java
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 [7], with the aspect weaving semantics proposed in Aspect Featherweight Generic Java [8]. 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 section 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) [6], namely RawFGJ [7] that provides a formal description of raw types in FGJ, and AFGJ [8], 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 $\mathcal{E}$ as an abbreviation for $e_1, e_2, \ldots, e_n$, for $n \geq 0$. The operation $\#(\mathcal{E})$ retrieves the length $n$ of the sequence $\mathcal{E}$. 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 $\overline{M}$.

The declaration `class C<\overline{XN}> \triangleleft N \{ \overline{FD} \ k \overline{MD} \}` defines a class $C$ with a list $\overline{XN}$ of type parameters, a parent type $N$, a list $\overline{FD}$ of field definitions, a constructor definition $k$, and a list $\overline{MD}$ of method definitions. The symbol $\triangleleft$ substitutes the
extends keyword for the sake of brevity. A type parameter definition $X \triangleleft N$ 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 cannot occur as the name of a declared class. Every declared class must specify a superclass, even when the superclass is Object. A field declaration $T \mathbf{f}$ simply associates type $T$ to a field name $\mathbf{f}$. Moreover, field definitions cannot be overwritten by subclasses, i.e. inheritance is conservative for fields (but not for methods). The constructor definition $\mathbf{C} (T \mathbf{g}, T \mathbf{f}) \{ \text{super}(\mathbf{g}) \text{this.f}=\mathbf{f} \}$ is the only mandatory part of the class body. The constructor parameters $\mathbf{T g}$ are used to initialize object fields defined in the super-class by means of $\text{super}(\mathbf{g})$, while the parameters $\mathbf{T f}$ are used to initialize the locally defined fields through $\text{this.f}=\mathbf{f}$. A method declaration $<X \triangleleft N> \mathbf{T m (T x)} \{ \mathbf{e} \}$ defines type parameters $X \triangleleft T$, a return type $T$, a method name $m$, method parameters $x$ and a method body $\mathbf{e}$.

An expression can be a variable reference $x$, a constructor invocation $\text{new } N(\mathbf{e})$, a field access $e.f$, a parametrized method call $e.m<T>(\mathbf{e})$, a raw method call $e.m(\mathbf{e})$, or advised versions of the two kinds of method invocation ($e.m[a]<T>(\mathbf{e})$ and $e.m[a](\mathbf{e})$, respectively). The advised methods are equipped with a list $[a]$ 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 $\text{advice a } T (T x) : \phi \{ e \}$ defines and advice of name
The signature $T \langle T \bar{x} \rangle$ 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 $\phi$ 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 $\phi$ 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 $\text{exe } T \ C.m(T)$ in the same sense as the execution pointcut of AspectJ, matching a method within class $C$, with name $m$, with parameters whose types are $T$, that returns a value of type $T$. In some places, like in method name or in type parameters, we can have the wildcard replacement $\ast$, 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 $C<T>$, is different from the correspondent raw type, denoted by $C$ alone, even if the list of type parameters is empty ($C<>$). The bottom type $\ast$ 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 $\Delta$, 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 $\Gamma$. A type environment can also have an special form of association, $T$ proceed($T$), 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 $\Delta \vdash Т ok$ denotes that the type $T$ is well-formed with relation to the type variable environment $\Delta$. It means that all type variables that occur in $T$ must respect the bounds defined by the type variable environment $\Delta$. 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.

\[
\begin{array}{c}
\Delta \vdash \text{Object ok} \quad \text{(WF-Object)} \\
\end{array}
\]
\[ X \in \text{dom}(\Delta) \]
\[ \Delta \vdash X \text{ ok} \quad \text{(WF-Var)} \]
\[ \begin{array}{c}
\text{class } C<X> \triangleleft N \{ \ldots \}
\end{array} \]
\[ \Delta \vdash \text{ok} \quad \text{(WF-Raw)} \]

The syntax \([T/X]N\) means the application of a sequence of type variable substitutions \(T_i/X_i\) in \(N\).

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

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

### 2.2.2. Subtype relation

The judgement \(\Delta \vdash T <: V\) says that type \(T\) is a subtype of \(V\) under type environment \(\Delta\). 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, \(T <: S \neq C<T> <: C<S>\). Considering a class definition like \(\text{class } C<X<:\text{Object}> \triangleleft N \{ \ldots \}\) the raw type \(C\) is the supertype of all its possible parametrizations, like \(C<\text{Boolean}>\), \(C<\text{String}>\) or \(C<\star>\). 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. \(C<\star> \not<:\ C<\text{String}>\). The rules that define the subtype relation are shown as follows.

\[ \Delta \vdash T <: T \quad \text{(S-Reflex)} \]
\[ \begin{array}{c}
\Delta \vdash S <: T \\
\Delta \vdash T <: U
\end{array} \]
\[ \Delta \vdash S <: U \quad \text{(S-Trans)} \]
\[ \Delta \vdash X <: \Delta(X) \quad \text{(S-Var)} \]
\[ \begin{array}{c}
\text{class } C<X> \triangleleft D \{ \ldots \}
\end{array} \]
\[ \Delta \vdash C<T> <: [T/X]D \quad \text{(S-Super)} \]

The relation \(\leq\), used by rule S-Raw, is the reflexive and transitive closure of the \(\text{extends}\) relation \(<:\) between classes. The rule S-CookedRaw says that the cooked type \(C<\text{Int}>\) 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 \textit{unsafe subtyping relation}, and is denoted as \(<:\gamma\). The rules for unsafe type judgements are provided as follows.

\[ \Delta \vdash C<T> <: C \quad \text{(S-CookedRaw)} \]
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 [7]).

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 choice 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 \(<Y\triangleleft P> T \rightarrow T\), 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.

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.

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 $|=\, (described in Section \ref{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.

$\varphi |= \text{exe } T\cdot\ast() \quad \varnothing \vdash T, T, C \, \text{ok}$

$\varnothing \vdash T.\ast() \; \text{target}, T \; \text{proceed}(\varnothing) \vdash e: T' \quad \varnothing \vdash T' <: T$

advice a \, T (T\cdot\ast()) : \varnothing (e) \, \text{ok}

$\varnothing \vdash T' ok$

$\varnothing \vdash T\cdot\ast()$ (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 $C.m \, \text{advisedby} \, a$ means that the advice $a$ intercepts method $m$ in class $C$.

$\text{advice } a \, T (T\cdot\ast()) : \varnothing (e) \quad \text{exe } T\cdot\ast() \, (T) \vdash \varnothing \quad \text{ok}$

$\text{exe } T\cdot\ast() \, (T) \vdash \varnothing \quad \text{ok}$

$\varnothing \vdash T\cdot\ast() \, (T) \vdash \varnothing \quad \text{ok}$

$\varnothing \vdash T\cdot\ast() \, (T) \vdash \varnothing \quad \text{ok}$

$\varnothing \vdash T\cdot\ast() \, (T) \vdash \varnothing \quad \text{ok}$

$\varnothing \vdash T\cdot\ast() \, (T) \vdash \varnothing \quad \text{ok}$

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

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 $\Delta$ as $|T|_\Delta$. This function is used by the method type fetching function $\text{mtypeadv}$. Both functions are defined as follows:

2. If the pointcut return or if the pointcut parameters are defined as raw types we must match it against erased method signatures. For example, if the pointcut is $\text{exe } \text{List } C.\ast()$, we must pick up all methods defined in class $C$ whose the erasure of return type is $\text{List}$ (e.g. $\text{List}, \text{List}<\text{Int}<>$, $\text{List}<X>$ and others). However, if the pointcut is using a fully defined type, $\text{exe List}<\text{Int}<> C.\ast()$ for example, we must perform an exact match (i.e. only methods with $\text{List}<\text{Int}<>$ as its return type must be picked up). To ensure this functionality we have created another auxiliary function, named $\text{matcheq}$.

$\text{matcheq}(T_0, T) \left \{ \begin{array}{ll} \text{head}(T_0) = T & \text{if } T \text{ is a raw type} \\ T_0 = T & \text{otherwise} \end{array} \right.$

2.2.5. Pointcut logic

The matching of execution points and advices is determined by a monotonic logic, initially proposed in \cite{8}. The rules that define the relation $|=\,$ are as follows.
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 $\ast$. Roughly speaking, the pointcut $\phi$ is less general (or more unified) than the pointcut $\psi$ if $\phi \models \psi$. The pointcut logic is used in the type system, when verifying 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 $R$-WEAVERT).

2.2.6. Expression typing

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

- The special class $\text{Object}$ must not occur in the class table, i.e. $\text{Object} \notin \overline{cd}$.
- The rules that deal with field and method lookup have special forms to deal with this class, since $\text{Object}$ 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 $C \in \overline{cd}$.
- The type $\ast$ does not occur anywhere in class or method definitions.
- For all $cd_i \in \overline{cd}$, $cd_i$ ok.

The sequence of advice declarations $\overline{ad}$ 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, $\ast$ does not occur in advice declarations and for all $ad_i \in \overline{ad}$, $ad_i$ ok. The judgement $\overline{ad}; \overline{cd} \vdash \text{ok}$ 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 $\overline{ad}$ 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 $\Delta; \Gamma \vdash e : T$ 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
2.3. Operational Semantics

The execution of Raw-AFGJ programs is given by means of standard structural operational semantics. The reduction judgement $\Delta; \Gamma \vdash e_1 \to e_2$ tells that program $\overline{cd} \overline{ad} e_1$ reduces to $\overline{cd} \overline{ad} e_2$ under environments $\Delta$ and $\Gamma$. We consider $\overline{cd}$ and $\overline{ad}$ 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 $\overline{ad}$. 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-ADVISER.
### Figure 3. Auxiliar definitions for type system rules

<table>
<thead>
<tr>
<th>Type Erasure</th>
<th>Cooked method type fetching</th>
</tr>
</thead>
</table>
| bound$_A$(X) = $\Delta$(X) | \[ \text{class } C<X\triangleright N ( \text{ld } k \text{ mad } ) \]
| bound$_A$(N) = N | \[ \langle \text{Cooked} \rangle R \triangleright m ( U X ) ( e ) \in \text{mad} \]
| head($C<T>$) = C | mtype$_R$(n, $C<T>$) = $\lfloor T/X \rfloor$ $\circ$ $\triangleleft R$ |
| head(C) = C | mtype$_R$(n, $C<T>$) = mtype(n, $T/X$) |
| $|T|_A = \text{head}(\text{bound}_A(T))$ | mtype$_R$(n, $C<T>$) = mtype$_R$(n, $T/X$) |

**Field types fetching**

\[ \text{fields}(\text{Object}) = \emptyset \]

**Constructor argument types fetching**

\[ \text{cargtype}(\text{Object}) = \emptyset \]

\[ \text{class } C<X\triangleright N ( \text{ld } k \text{ mad } ) \text{ fields}(\lfloor T/X\rfloor N) = Tn \text{ cargtype}(C<T>) = Tn, \lfloor T/X\rfloor \text{ld} \]

\[ \text{class } C<X\triangleright N ( \text{ld } k \text{ mad } ) \text{ fields}(N) = Tn \text{ cargtype}(C) = Tn, \text{ld} \times N \]

\[ \text{class } C<X\triangleright N ( \text{ld } k \text{ mad } ) \text{ cargtype}(\lfloor T/X\rfloor N) = Tn \text{ cargtype}(C<T>) = Tn, \lfloor T/X\rfloor \text{ld} \]

\[ \text{class } C<X\triangleright N ( \text{ld } k \text{ mad } ) \text{ cargtype}(N) = Tn \text{ cargtype(C)} = Tn, \text{ld} \times N \]

\[ \text{warning if } Tn, Tn_A \neq Tn, Tn \]

\[ \text{warning if } Tn, Tn_A \neq Tn, Tn \]

\[ \text{mtype}_{R}(n, C<T>) = \text{mtype}_{R}(n, T/X) \]

\[ \text{mtype}_{R}(n, C<T>) = \text{mtype}_{R}(n, T/X) \]

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.

\[ [v.m[\{x\}(3)]/\text{proceed}] \text{proceed}(\tau) = v.m[\{x\}(\tau, \{x\})] \]

\[ [v.m[t\{x\}(3)]/\text{proceed}] \text{proceed}(\tau) = v.m[t\{x\}[\{x\}](\tau, \{x\})] \]

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.

### 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.\[8\]. 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 contain type variables and then tries an exact match over this signature. If the method contains type variables, type erasure is applied and the matching is done against its result\(^1\). In Raw-AFGJ, this behavior is modelled by the \(\llbracket\Delta\rrbracket\), 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 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 parametrization 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 no-

\(^1\)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.
table 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 $\emptyset;\emptyset \vdash cd \rightarrow \emptyset;\emptyset \vdash ok$ and $\neg\text{warning}$, then $e$ is a value or $\emptyset;\emptyset \vdash cd \rightarrow cd; cd e'$. 

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

Both proofs follow the standard syntactic soundness technique and are available in [http://www.inf.ufrgs.br/~rma/rawafgj](http://www.inf.ufrgs.br/~rma/rawafgj).

### 4. Related work

Featherweight Java (FJ), proposed by Igarashi et al. [6], 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) [6] which supports parametric polymorphism; and (2) Raw Featherweight Generic Java (Raw-FGJ) [7] that extends FGJ with raw types.

Recently, some aspect-oriented calculi were proposed to study the influence of aspects over object-oriented languages [3, 8], and also to study aspects in isolation [2, 5]. In Jagadeesan et al. work [8] 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.

Fraine et al. [4] have presented an analysis of the type systems of current aspect oriented languages and how they can cause type errors that remain undetected at compile time. 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. We would like to acknowledge CAPES for the given support.

References