UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FERNANDO BARDEN RUBBO

Inference Rules for Generic Code Migration of Aspect-Oriented Programs

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Daltro José Nunes
Advisor

Porto Alegre, August 2009

CIP – CATALOGING-IN-PUBLICATION

Rubbo, Fernando Barden

Inference Rules for Generic Code Migration of Aspect-Oriented Programs / Fernando Barden Rubbo. – Porto Alegre: PPGC da UFRGS, 2009.

92 p.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2009. Advisor: Daltro José Nunes.

1. Generics. 2. Refactoring. 3. Aspect-oriented. I. Nunes, Daltro José. II. Title.

TABLE OF CONTENTS

LIST OF ACRONYMS

GJ	*Generic Java*
GA	*Generic AspectJ*
OO	*Object-Oriented*
AO	*Aspect-Oriented*
JVM	*Java Virtual Machine*
AFGJ	*Aspect Featherweight Generic Java*
API	*Application Program Interface*
JDK	*Java Development Kit*
AST	*Abstract Syntax Tree*
AOP	*Aspect Oriented Programming*
IDE	*Integrated Development Environment*
CU	*Compilation Unit*

LIST OF LISTINGS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The latest versions of AspectJ – the most popular aspect oriented extension for Java – must cope with complex changes that occurred in the Java type system, specially with the parametric polymorphism which aims to improve the type safety and the readability of the source code. However, for legacy and non-generic constructions to take advantage of this pervasive feature, they must be migrated to explicitly supply actual type parameters in both declarations and instantiations of generic classes. Even though the type systems of Java and AspectJ were designed to support this kind of migration in a gradual way, this process is somewhat complex and error prone. The reason behind this assertion is that actual type parameters must be inferred to remove as much unsafe downcasts as possible without affecting the original semantics of the program. Therefore, tools are essential to minimize the effort of a manual application of the refactoring steps and to prevent the introduction of new errors. Since current automated solutions focus only on Java programs, they do not consider the use of aspects to encapsulate crosscutting concerns. Thus, this dissertation proposes a novel collection of inference rules to derive type constraints for the polymorphic version of AspectJ. These rules were used together with an existing generic migration algorithm to enable the conversion of non-generic legacy code to add actual type parameters in both Java and AspectJ languages.

Keywords: Generics, refactoring, aspect-oriented.

1 INTRODUCTION

The Java language evolved and came to include, in its version 1.5, parametric polymorphism[1] for classes, interfaces and methods (BRACHA et al., 1998). In Java, the parametric polymorphism, also known as Generic Java (GJ), is mainly used to provide better compile-time type checking and to improve the source code readability – since types help document program's functionalities and programmer's intentions. Naturally, such improvements raised concerns about compatibility with legacy programs in both source and bytecode levels.

The source level compatibility issue was solved by expanding the type system to support parameterless instantiation of parametrized types, the so called *raw types* (BRACHA et al., 1998). The raw type design was created to provide a convenient support for non-generic legacy code through the generic software evolution. In other words, legacy classes can be gradually transformed into polymorphic versions without imposing dependencies between software modules developed independently.

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 bytecode very close to the older non-generic one.

Using these strategies, the Generic Java platform does not provide a full and sound reification[2] of the generic type system (GOSLING et al., 2005). But, it provides a good support for evolving existing non-generic code to take advantage of generic types. The simple example shown below depicts this evolution:

```
Non-Generic Code

1   List l = new ArrayList();
2   l.add(new Integer(1));
3   String s = (String)l.get(0); // well-typed
```

```
Generic Code

1   List<Integer> l = new ArrayList<Integer>();
2   l.add(new Integer(1));
3   String s = (String)l.get(0); // ill-typed
```

This example presents a non-generic legacy code on the left side and the equivalent generic version on the right side. On the left side, the *List* declaration and the *ArrayList* instantiation (line 1) were written in a way that, though old-fashioned, is still well-typed in the GJ type system. On the other hand, as it can be seen in the last line of the same example, the legacy non-generic support allows the programmer to write code that may cause type conversion errors during the program execution (note that, in this line, an object of the type *Integer* is being cast to *String*). By

[1] According to Cardelli and Wegner (1985), parametric polymorphism – also known as generics – is obtained when an object or a function works uniformly on a set of types, which is normally provided through explicit or implicit parameters.

[2] In Java programming language there exist "reifiable types". These types are not affected by type erasure, and therefore, are completely available at runtime.

providing all type parameters for generic classes and removing redundant downcasts (as demonstrated in the right side), the Java compiler is capable of identifying more type errors at compile-time than the legacy non-generic support (see the line 3 on the right side).

The AspectJ language (KICZALES et al., 2001) is a super set of Java that adds some aspect-oriented (AO) abstractions, such as: pointcuts, advices, inter-type declarations and others. Following Java 1.5, AspectJ has also added support for parametric polymorphism. As a result, parameterized types may be freely used within aspects. Consider the following piece of code:

Non-Generic Code

```
1    class C{
2      void m(List l){
3        Integer i = (Integer)l.get(0);
4        // do something
5      }
6    }
7    aspect A{
8      before(List l):
9          execution(void C.m(..)) && args(l) {
10       Integer i = (Integer)l.get(0);
11       // do something
12     }
13   }
14
15   ..
16   List l = new ArrayList();
17   l.add(new Integer(1));
18   new C().m(l);
```

Generic Code

```
1    class C{
2      void m(List<Integer> l){
3        Integer i = (Integer)l.get(0);
4        // do something
5      }
6    }
7    aspect A{
8      before(List<Integer> l):
9          execution(void C.m(..)) && args(l) {
10       Integer i = (Integer)l.get(0);
11       // do something
12     }
13   }
14
15   ..
16   List<Integer> l = new ArrayList<Integer>();
17   l.add(new Integer(1));
18   new C().m(l);
```

The above example defines a *before* advice (line 8) that will be triggered by the execution of method *m* (line 2). Note that, in the left side, a cast is required every time an element is retrieved from objects of the type *List* (lines 3 and 10). As it was seen previously, this kind of construct is not safe and may cause errors at runtime. Conversely, on the right side, actual type parameter annotations were included in all declarations of *List* (lines 2, 8 and 16) and in the instantiation of *ArrayList* (line 16). Because of that, the casts became redundant and then were removed (lines 3 and 10). This generic version of the code guarantees that the program will be type safe at compile-time, even in the presence of aspect *A*.

It is important to emphasize that the parametric polymorphism of both Java and AspectJ improves the type safety and the expressiveness of the source code. However, for non-generic code to take advantage of this pervasive feature, it must be migrated to explicitly supply actual type parameters in both declarations and instantiations of generic classes.

Even though the type systems of both languages were designed to support such migration, this process – when performed manually – can be tedious, time consuming and error prone (DINCKLAGE; DIWAN, 2004; MUNSIL, 2004; DONOVAN et al., 2004). The reason behind this assertion is that actual type parameters must be inferred to remove as much unsafe downcasts as possible without affecting the original semantics of the program. This is known in the generic migration literature as the *instantiation problem* (DONOVAN et al., 2004).

There are several approaches that aim to help solving this problem (DONOVAN et al., 2004; DINCKLAGE; DIWAN, 2004; FUHRER et al., 2005; CRACIUN et al., 2009). In general, they analyze the source code looking for gaps of type information and then rewrite a new semantically equivalent generic version of the program, in

which actual type parameters are automatically inserted and redundant downcasts are removed.

These solutions have been successfully used for migrating pure object-oriented (OO) software, but they are not able to ensure that the migration will be successful in the presence of aspects. There are several subtleties involving both AspectJ's type system, inter-type declarations, parent declarations and pointcut expressions that make this task more complex and error prone. Therefore, approaches focused on the aspect-oriented context must deal with the following:

- **Poor inference:** aspect-oriented constructs must be considered during the program transformation since they may change the inference results. Without analyzing such constructs, the inferred types may not be as good as expected and the code inside aspects will not be improved;

- **Ill-typing:** since inter-type and parent declarations are implicitly woven into the application, actual type parameters inferred may become ill-typed when the weaver adapts the structure or the hierarchy of a class;

- **Pointcut's fragility:** pointcuts can match elements across the entire base code. Thus, reasoning about the correctness of a pointcut requires a deep understanding of the internal structure of the software (YE; VOLDER, 2008; WLOKA et al., 2008). Therefore, the task of adding actual type parameters to raw declarations and instantiations, for example, can be troublesome;

- **Inference of wildcards:** since aspects are specifically designed to deal with crosscutting concerns, the use of wildcards in some declarations (such as, *args*(..) pointcut primitives and *after*() *returning*(..) advices) allows the removal of more unsafe downcasts than the use of a specific type;

- **An unsafe type system:** there are AspectJ's constructs that are not type safe even with the use of parameterized types. This happens because there are some typing rules that severely restrict advice definitions obligating the matching against type variables to happen against their erasure.

This dissertation presents a novel collection of type constraint rules for the polymorphic version of AspectJ. These rules were used together with an existing generic migration algorithm to address the instantiation problem in the AO context, enabling the conversion of non-generic legacy code to add actual type parameters in both Java and AspectJ languages.

Even though the idea of extending existing approaches to perform this kind of migration in the AO context has been previously suggested (HANNEMANN, 2006), a concrete proposal has not been found. Therefore, it is important to highlight what the main distinctive characteristics of this work are:

- **The improvement of existing object-oriented support**. The aspect-aware generic code migration proposed in this dissertation generates actual type parameters for several OO constructions that were not considered in the original solution (FUHRER et al., 2005): (i) raw types used in the signature of methods declared into generic classes; (ii) calls to methods defined in the super generic class; and (iii) subclasses of generic classes considering the case where no types can be inferred due to GJ limitation;

- **The inference of wildcards**. The solution proposed in this work generates, for both Java and AspectJ constructs, wildcard constraints when applicable;
- **The aspect-oriented support**. The proposed solution also considers the use of aspects to encapsulate crosscutting concerns. This enables existing AspectJ code to take advantage of the polymorphic support provided by the platform;
- **The aspect-aware type constraint framework**. The aspect-aware type constraint framework was used in this work for generic migration. However, there are other refactorings, such as generalizations (TIP et al., 2003), parameterization of non-generic classes (KIEŻUN et al., 2007) and the customization of container classes (SUTTER; DOLBY, 2004), that can also be extended to transform AO programs using the type constraint framework proposed in this study.

The remainder of this dissertation is organized as follows. Chapter 2 provides an overview the Java and the AspectJ generic type systems including their main design decisions. Chapter 3 presents the inference rules for deriving type constraints. Chapter 4 describes the use of these rules in an existing generic migration algorithm and discusses a couple of examples which claim for migration to take advantage of generics. Chapter 5 includes some concluding remarks and, finally, Appendix A presents the detailed application of the algorithm steps in some examples.

2 BACKGROUND

This chapter describes some of the background areas which help understand the main issues of this dissertation. In summary, it overviews how Generic Java (GJ) and Generic AspectJ (GA) type systems were designed; the main advantages of performing generic code migration; the main features that may be employed during this kind of migration and; the limitations imposed by some design decisions, such as raw types and type erasure.

These and other relevant information, including syntax and semantics of GJ and GA, are informally discussed in the remainder of the chapter. For more details, the reader is encouraged to take a look at the GJ foundation (ODERSKY; WADLER, 1997; BRACHA et al., 1998; IGARASHI et al., 2001a,b; GOSLING et al., 2005) and at the GA basis (ASPECTJ, 2005; JAGADEESAN et al., 2006; RUBBO et al., 2008; AVGUSTINOV et al., 2007).

2.1 Generic Java

According to Gosling *et al.* (2005), Generic Java – the parametric polymorphism for Java – is a key feature that allows us to take advantage of homogeneous data structure libraries in a flexible way. Using the GJ support, it is possible to specify and utilize generic classes, interfaces and methods to obtain a better compile-time type checking and to improve the source code readability (since types help document program's functionalities and programmer's intentions).

2.1.1 Generic Classes

In order to declare a generic class, it is necessary to define its formal type parameters just after the class name. These type parameters can be used as normal types[1] in non-static declarations within the class. Syntactically, if *C* represents the class name, the formal type parameters must follow *C*'s declaration in a list separated by commas and between ⟨ and ⟩. Then, upper bounds may be specified for each type parameter using the keyword *extends*. Moreover, in cases where multiple bounds are required, they must be separated by &. If no bound information is provided, *Object* is assumed.

Listing 2.1: Generic classes

```
1  class C1<F1> {..}
2  class C2<F2 extends Number, F3> {..}
3  class C3<F4 extends MyClass & MyInterface>{..}
```

[1] There exists some limitations that will be briefly discussed in Section 2.1.3.

In Listing 2.1, there are three generic classes that attempt to demonstrate different possibilities for class parameterization. The first class, $C1$, was declared with one formal type parameter, $F1$, which defines implicitly *Object* as its upper bound. Class $C2$ specifies two formal type parameters, $F2$ and $F3$, which define respectively *Number* and *Object* as their bounds. Finally, class $C3$ specifies only one formal type parameter, $F4$, with two bounds (*MyClass* and *MyInterface*). In the last case, if both *MyClass* and *MyInterface* define a method with the same signature, any reference to that method will be ambiguous and then will cause an error at compile-time.

Generic Java also allows a given class C to inherit from a generic class B. In this kind of inheritance, if there are formal type parameters specified in C, the variables bounded to those parameters may be used as real types during the instantiation of its super class B. As an example, consider the fragment of code depicted in Listing 2.2.

Listing 2.2: Generic classes inheritance

```
1  class C4<F1 extends Number> {..}
2  class C5<F2 extends Number> extends C4<F2> {..}
3  class C6 extends C4<Integer> {..}
```

In this example, class $C4$ declares a formal type parameter, $F1$, with *Number* as its upper bound. Class $C5$ declares the formal type $F2$ (also bounded to *Number*) and passes it as an actual type to its super class $C4$. At last, $C6$ (which does not have type parameters) always passes *Integer* to its super class.

Once a given generic class C defines a list of formal type parameters, $\langle F_1,..,F_n\rangle$, its declarations and/or instantiations require that all actual type parameters, $\langle A_1,..,A_n\rangle$, be informed. Each actual type, A_j (where $1 \leq j \leq n$), must respect the bounds defined in the corresponding formal declaration, F_j. Thus, looking at Listing 2.2, it is possible to see that $C4$ and $C5$ instantiations must inform one of the valid *Number*'s subtypes (i.e. *Float*, *Integer* and etc.) as the actual type parameter. On the other hand, for the instantiation of $C6$ no type parameter has to be informed because it is not a generic class. Consider the following example:

Listing 2.3: Generic class instantiation

```
1  C5<Float> a1 = new C5<Float>();
2  C4<Integer> a2 = new C6();
```

In the first line, the variable $a1$ is both declared and instantiated as an object whose type is $C5\langle Float\rangle$. In line 2, variable $a2$ is being declared as $C4\langle Integer\rangle$ and instantiated as $C6$. Note that, even though $C6$ has no type parameters, the statement is considered well-typed because the declaration of this class (see Listing 2.2) is passing *Integer* as the actual type parameter to its super class $C4$.

2.1.2 Invariant Subtyping

Different instances of generic classes are not related by "normal" type hierarchy, in which a given object is an instance of *Number* if it was created from one of the *Number*'s subtypes (such as *Integer*, *Float* and etc). Unlike arrays, generic types are not co-variant[2]. This means, in a scenario where $C5$ extends $C4$ (see the classes

[2]In the current Java specification (GOSLING et al., 2005), arrays of generic types (such as, *new ArrayList*$\langle$*String*$\rangle$[0]) are not allowed.

declaration in Listing 2.2), $C5\langle A1\rangle$ is a subtype of $C4\langle A2\rangle$ if and only if $A1$ is equal to $A2$.

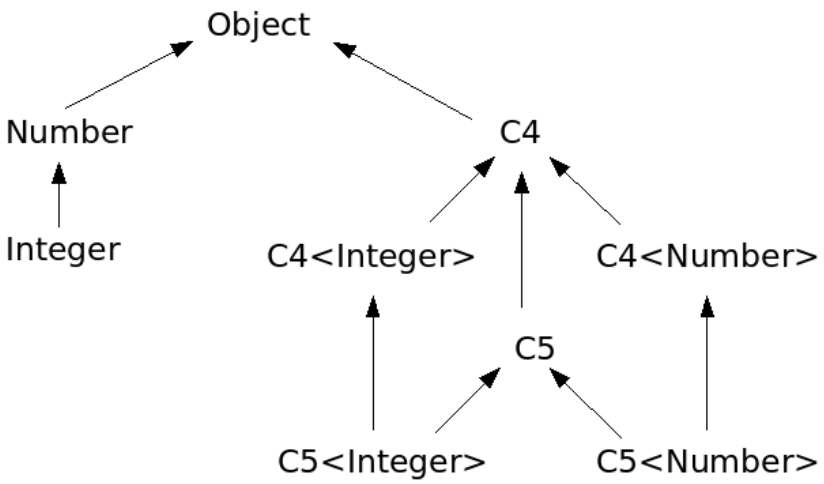

Figure 2.1: Example of Generic Java hierarchy

Figure 2.1 (extracted and modified from (DONOVAN et al., 2004)) depicts an example of the generic type hierarchy in the Java language. In this example, it is possible to notice that $C5\langle Integer\rangle$ is not a valid subtype of $C4\langle Number\rangle$ even knowing that *Integer* is a subtype of *Number*. As a rule, it can be assumed that whenever a given class C is a subtype of another class B all type parameters defined in this hierarchy must be invariant. That is the reason why $C5\langle Integer\rangle$ is a valid subtype of $C4\langle Integer\rangle$.

2.1.3 Raw Type and Type Erasure

The GJ type system assumes that a declaration or an instantiation of a generic class passing no actual type parameters is a *raw type*. Raw types were created to ensure that legacy "monomorphic"[3] code could coexist with the newer polymorphic version without the necessity of modifying existing source codes. This design provides a convenient support for non-generic classes through the generic software evolution (BRACHA et al., 1998; IGARASHI et al., 2001b). In other words, legacy classes can gradually be transformed into polymorphic versions without imposing dependencies between independently developed modules.

In short, a raw type is a generic type in which each actual type parameter may be faced as an existential type, such as $C\langle \exists x \ . \ x \leq Object\rangle$[4]. Since always there is a valid x that is a subtype of *Object*, anyone can instantiate a generic class C without informing the actual type parameters (e.g., *new* $C()$). Based on Figure 2.1, it is possible to create some examples to demonstrate the raw type usage:

Listing 2.4: Examples of raw type usage

```
1 C4 a1 = new C5();
2 C4 a2 = new C5<Integer>();
3 C4<Integer> a3 = a2;
4 C4<Float> a4 = a2;
```

[3] In this work the term "monomorphic" is used as a synonym for non-generic.

[4] At this point, relation $\leq$ is used to express Java subtype relationship.

In line 1, the variable *a1* is a raw type because no actual type parameters are provided either in *C4*'s declaration or in *C5*'s instantiation. Similarly, in line 2, *a2* is also considered a raw type because *C4*'s declaration does not inform the required type parameter. Conversely, the variables *a3* and *a4* (in line 3 and 4, respectively) are cooked types[5] since their declarations inform what are the type parameters being used.

If on one hand raw types can accommodate interaction with legacy code, on the other hand they obligate the type system to have some type rules that are deliberately unsound (IGARASHI et al., 2001b). To illustrate such circumstance, take a look at the assignment declared in line 4 of Listing 2.4. In this case, an object of type *C5*⟨*Integer*⟩ is being assigned to a variable declared as *C4*⟨*Float*⟩. As it was discussed previously, subtyping in GJ is invariant, hence this construction should be an ill-typed statement. However, because of the raw type design, the assignment is considered well-typed at compile-time but will cause a type conversion error at runtime.

In order to make Java programs type safe, it is strongly recommended that raw types be refactored to inform required type parameters (GOSLING et al., 2005; DONOVAN et al., 2004; KIEŻUN et al., 2005; DINCKLAGE; DIWAN, 2004). In an effort to simplify this task, the Java compiler warns the programmer saying that some expressions can not be fully type checked, and therefore, they may cause errors during the program execution.

While raw types provide a way of ensuring backward compatibility in the source code, the type erasure design aims to solve the compatibility issue in the bytecode. This means that, after type checking, the Java compiler erases all type parameter information and inserts the necessary downcasts in an effort to create a semantically equivalent bytecode that is very close to the legacy non-generic version.

Due to both raw type and type erasure designs, information about type parameters are not available at runtime and, consequently, operations such as *instanceof* are not able to check generic data in the current GJ specification (GOSLING et al., 2005). In order to make some generic information available through reflection, the specification defines a new bytecode element called *Signature*. This element gives enough support to retrieve *declared* type parameters of attributes, methods, classes and interfaces. However, it is important to highlight that it is still not possible to retrieve the *actual* type parameters since the Java Virtual Machine (JVM) does not keep track of this information.

Listing 2.5: Bytecode of a method using a parameterized type

```
1  // Method descriptor #17 (Ljava/util/List)V
2  // Signature: (Ljava/util/List<Ljava/lang/Integer>)V
3  // Stack: 0, Locals: 2
4  void m(java.util.List e);
5  ...
```

In order to exemplify the result of the type erasure process, it is shown in Listing 2.5 the bytecode of a method called *m*. As it can be seen in line 4, this method seems to be declaring a formal parameter *e* with type *java.util.List*. However, the truthful declared type information of this method is only available in the *Signature* element depicted in line 2. Looking carefully at this line, it is possible to infer that *m* is

[5] A cooked (or also known as parameterized) type, is a fully defined type. Which means that the whole type information is provided for a given generic class.

actually declaring (in the source code) its parameter *e* as *java.lang.List*⟨*Integer*⟩ instead of the raw *java.lang.List*.

2.1.4 Generic Methods

Likewise classes, a list of formal type parameters can also be associated with methods. Such parameter list must be specified at the beginning of the generic method[6] signature following the list of modifiers. For example:

Listing 2.6: Generic methods

```
1  public <F1> F1 m1(F1 arr) {..}
2  public <F2, F3 extends F2> void m2 (F2 a, F3 b){..}
```

In line 1, the method declaration specifies an unbounded formal type parameter $F1$ which is being used to express type interdependency between the parameter and the return. Similarly, in line 2, the unbounded type parameter $F2$ is being used as the upper bound for the type parameter $F3$. In Java, it is illegal to declare two methods with the same signature in the same class. This definition is also extended to generic methods, hence if two methods have the same name and the same number of arguments, at least one type parameter must have different bounds. For example:

Listing 2.7: Generic methods with the same name

```
1  <F1, F2 extends F1> void m (F1 a, F2 b){...}
2  <F3 extends Number, F4> void m (F3 a, F4 b){...}
```

The two generic methods declared in Listing 2.7 have the same signature but different limits (upper bounds) for their type variables. This can be noticed looking at the first formal type parameter of each declaration[7]. While the first method (line 1) bounds *Object* to $F1$, the second (line 2) bounds *Number* to $F3$.

It is also important to mention that there is no need to use a special syntax to invoke generic methods. The Java compiler is smart enough to implicitly infer the actual type parameters for each invocation. Therefore, generic methods can be called exactly in the same way as non-generic ones – this means that no type parameter has to be informed[8].

2.1.5 Wildcards

Wildcards (TORGERSEN et al., 2004) are special constructs that may be used as type parameters in declarations of generic classes to provide variance to the Java invariant subtyping. Usually they are useful in situations where only partial knowledge about the type parameter is required. Wildcard types were included into the GJ type system because it augments the expressiveness and the flexibility of generic declarations (GOSLING et al., 2005).

As with raw types, wildcards can also be considered as an existential type. The main difference between the two is that for wildcards it is possible to specify upper and lower bounds (? *extends* C and ? *super* C, respectively). If no bounds are defined,

[6] A generic method is a method which has type parameters declared in its signature.

[7] Note that different type variable names ($F1$ and $F3$) were used to disambiguate. However, the same name could be used since the variables were declared in different scopes.

[8] The GJ specification defines a special syntax to explicitly inform the type parameter list during generic method invocation. However, since this usage is rare in practice this dissertation does not go into details about this subject.

Object is assumed as the upper bound. Take the following method declaration as an example:

Listing 2.8: Example of wildcard usage

```
1  void m(Collection<? extends Number> coll) {
2      for (Number n : coll) {
3          ...
4      }
5  }
```

In this example, the method parameter *Collection⟨? extends Number⟩ coll* specifies that the caller must pass a collection of something that extends *Number* (i.e. *Collection⟨∃x . x ≤ Number⟩*). Note that a similar result can be obtained using a generic method. But, according to Gosling *et al.* (2005), generic methods should be used only to express type interdependency between arguments, exceptions and/or return. In the lack of this interdependency, wildcards should be preferred.

Listing 2.9: Example of *java.util.Collection*

```
1  public interface Collection<E> extends Iterable<E>{
2      boolean addAll(Collection<? extends E> c);
3      ...
4  }
```

A good example of upper bound wildcard usage is the method *addAll* defined in *java.util.Collection* (see Listing 2.9). In this case, the wildcard declared as *? extends E* specifies that the method *addAll* accepts collections of objects that extends the type parameter *E*, which is informed during the instantiation of the class. To understand the power of this wildcard usage take a look at the following example:

Listing 2.10: Example of *Collection.addAll(..)* usage

```
1  Collection<Integer> cInt = new ArrayList<Integer>();
2  ...
3  Collection<Number> cNumber = new ArrayList<Number>();
4  cNumber.addAll( cInt );
```

In line 4, all elements of *cInt* are being added into *cNumber* (line 4), even knowing that *cNumber* and *cInteger* are collections parameterized with different types (*Number* and *Integer*, respectively). It is important to highlight that, if the parameter of the *addAll* method were declared as *Collection⟨E⟩* instead of *Collection⟨? extends E⟩*, the operation shown in line 4 would cause a compilation error because the code would violate the invariant subtyping described in Section 2.1.2.

2.2 Generic AspectJ

AspectJ is an aspect-oriented (AO) language (KICZALES et al., 2001) based on Java. Besides the usual OO constructs (classes, methods, fields and etc), the language provides abstractions related to aspects implementation, such as: pointcuts, joinpoints, advices and inter-type declarations.

In the AspectJ terminology, the specification of what to do is called advice and the specification of when to do is called pointcut. In summary, pointcuts group joinpoints – which are well-defined points in the execution flow of a program – by the definition of a predicate that, whenever satisfied, causes the associated advices to be executed. State or behavior can also be introduced into existing components

through inter-type declarations. A collection of pointcuts, advices and inter-type declarations organized to perform a coherent task is called aspect[9].

Since AspectJ is a superset of Java, it is natural that the language has followed the GJ design adding support for parametric polymorphism. As a result, parameterized types can freely be used within aspects, including pointcut expressions and inter-type declarations.

2.2.1 The Polymorphic Support

AspectJ 5 provides full support for all of the Java 5 language features, including generic types (ASPECTJ, 2005). Consequently, every syntax, semantics and design decisions discussed so far also apply to AspectJ.

In summary, the AspectJ's polymorphic support – also known as Generic AspectJ – is an extension of the Generic Java with the addition of the possibility of using parameterized types within aspect members. However, due to some GJ design decisions (see Section 2.1.3), the AspectJ's type system was obliged to cope with the lack of type information during the weaving-time – which takes place just after the erasure process.

With regards to pointcut expressions, both raw and cooked types may be used. However, the target matching class must be a raw type, otherwise the construction will be ill-defined. For instance, consider the following advice declaration:

Listing 2.11: Ill-defined pointcut expression

```
1  before(): execution(void C<String>.m()){
2      System.out.println("Executing foo in C<String>");
3  }
```

The advice shown in Listing 2.11 aims to print a message before each execution of method *m*; but only when it is called from an object of the type $C\langle String\rangle$. Although this seems to be a valid declaration, it is considered ill-typed by the AspectJ type system. This happens because at weaving-time there is no way to distinguish between $C\langle String\rangle$ and $C\langle Integer\rangle$, for example. This limitation is imposed by the erasure design, and therefore, AspectJ only accepts raw types as valid targets in pointcut expressions.

Besides that, there are other cases also affected by the erasure design. Usually they are primitive pointcut constructions, such as *this*(..) and *target*(..), which depend on the runtime type of their arguments to define if a determined advice matches or not a given joinpoint. In these cases, AspectJ's type system also disallow the use of cooked types.

2.2.2 Matching Generic Types

The matching mechanism has also been adapted to support the AspectJ's polymorphic type system. Likewise Java, it uses raw types to maintain the source level backward compatibility. Meaning that those types, when used in pointcut expressions and type patterns, ensure that pointcuts already written, in existing non-generic code, will continue to work as expected when it is migrated to a new generic version.

[9]The text comprising the remainder of this dissertation is assuming that the reader has, at least, a basic knowledge about the aspect-oriented definitions used by AspectJ.

In contrast, cooked types, when used in pointcut's signature, aim to restrict advice matchings. This is achieved by specifying a parameterized type at the appropriate point in the signature pattern. To illustrate how these types can be used, consider the following example:

Listing 2.12: Using parameterized types in pointcuts

```
1  class C{
2      void m1(List<Integer> e){..} // matches
3      void m2(List<Double> e){..}  // do not match
4  }
5
6  ...
7  before(): execution(void *(List<Integer>)){..}
```

The advice depicted in line 7 of the above listing declares a pointcut that matches any method which has a unique parameter of the type *List⟨Integer⟩*. Note that, in this example the advice is matching only method *m*1 (declared in line 2); but, if a raw *List* were used in the signature pattern instead of the cooked *List⟨Integer⟩*, both methods *m*1 and *m*2 would be matched.

It is important to understand that the matching of parameterized types is only possible because of the new bytecode element called *Signature*. As it was discussed in Section 2.1.3, this element stores the full *declared* type information in the bytecode. Then, AspectJ uses it during the weaving process to decide which members have a match.

When wildcards come to pointcut's signature, the parameterized type using a generic wildcard (*List⟨?⟩*, for example) is a distinct type. This means that *List⟨?⟩* is a very different type when compared with *List⟨String⟩*, even though a variable of type *List⟨String⟩* can be assigned to a variable of type *List⟨?⟩*.

Additionally, as it was seen previously, some primitive pointcuts (such as *this*(..) and *target*(..)) do not support parameterized types. But, cooked types may be used in conjunction with *args*(..) primitive and *after*() *returning*(..) advices. According to AspectJ discussion forum[10], these constructions should work like Java casting conversion (GOSLING et al., 2005) with a subtle difference: instead of throwing a *ClassCastException* when an invalid cast is found, it should not execute the advice. To illustrate the usage of parameterized types along with *args*(..) constructions, take a look at the following example:

Listing 2.13: *Args* usage example

```
1  public class A<E extends Number> {
2      void m1(List<Integer> e){}           // match
3      void m2(List<? extends Number> e){} // unsafe match
4      void m3(List<Number> e){}            // does not match
5  }
6
7  ..
8  before() : execution(void *(..)) && args(List<Integer>){}
```

Although the signature matching part (*execution*(*void* *(..))) of the pointcut expression shown in line 8 aims to match any method whose return is *void*, the primitive *args*(*List⟨Integer⟩*) tells the weaver that only methods with one parameter that can be cast to *List⟨Integer⟩* must have a match. That is the reason why methods *m*1 and *m*2

[10]Differently than Java, AspectJ does not have a semi-formal specification. Moreover, its main generics documentation (ASPECTJ, 2005) is incomplete and ambiguous. That is why the following thread was started: `http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg09949.html`.

(lines 2 and 3, respectively) have a match and method *m*3 (lines 4) does not. It is important to highlight that in the case of method *m*2, the weaver marks the advice with an appropriate warning; since casting *List*⟨? *extends Number*⟩ to *List*⟨*Integer*⟩ is an insecure conversion.

In addition to these generic peculiarities, the matching is also affected by type erasure. As it was already discussed, no generic information remains in the bytecode and, therefore, AspectJ is not capable to match against type variables. Hence, joinpoints that have this kind of variables as part of their signature are matched by their erasure. For instance, consider the following declarations:

Listing 2.14: Matching against type variables

```
1  class C<E extends Number>{
2      void m1(E e){..}
3      void m2(List<E> e){..}
4  }
5
6  ...
7  before(): execution(void C.m1(E)){..}          // does not match
8  before(): execution(void C.m1(Number)){..}     // matches m1
9  before(): execution(void C.m2(List<E>)){..}    // does not match
10 before(): execution(void C.m2(List)){..}       // matches m2
```

The advices declared at lines 7 and 9 will not match any method because AspectJ does not allow the matching against type variable references. On the other hand, since type variables must be matched by their erasure, the advices declared at lines 8 and 10 will match methods *m*1 and *m*2, respectively.

2.2.3 Inter-type and Parent Declarations

Inter-type and parent declarations introduce state or behavior in an existing class, aspect or interface. While inter-types allow us to define methods and attributes that will be added to a specified class/aspect during the weaving process, declare parents constructs augment the type hierarchy as long as it results in a well-formed structure in accordance with Java's subtyping rules. It is important to highlight that those declarations can adapt class structures without modifying the adapted implementation module. However, the adapted code can be inconsistent if it is not combined with the aspect module that makes its structure well-formed. For example:

Listing 2.15: Inter-type declaration example

```
1  class C{
2      void m1(List l){
3          m2(l);
4      }
5  }
6  aspect A{
7      void C.m2(List l){
8          ...
9      }
10 }
```

Class *C* is well-formed only in the presence of aspect *A*, since *C* depends on *A* to declare method *m*2. Although this kind of definition provides a modular way to organize the code that deals with crosscutting concern, they can make it difficult to reason about the behavior of the program. Those constructions, also make it difficult to infer the right actual type parameter during the generic code migration because the aspect weaving, in the AspectJ platform, takes place only in a bytecode

level.

2.2.4 Generic Aspect

Similarly to classes, a list of formal type parameters can also be specified for abstract aspects. These type parameters must be declared just after the aspect name and they may be used in the same way that those declared in generic classes. Since AspectJ does not allow the weaving of abstract aspects, only concrete constructs with all their type parameters fully defined are actually weaved. Consider the following example:

Listing 2.16: Generic aspect

```
1  class C{
2      void m1(List<Integer> l){..}
3  }
4  abstract aspect A<E extends Number>{
5      before(): execution(void C.m1(List<E>)) // does not match
6      {..}
7  }
```

Listing 2.16 presents a generic aspect (line 4) which declares a type parameter *E* bounded to *Number*. Even though *Integer* extends *Number*, the advice defined at line 5 does not match method *m1* (declared at line 2). This happens because AspectJ requires that all type parameters be informed in order to weave a generic aspect[11]. Thus, this kind of construction is useless if no concrete sub-aspect is defined.

Listing 2.17: Sub-aspects of generic aspect

```
1  aspect A2 extends A<Integer>{}
```

To exemplify the utility of a generic aspect, the above code defines a concrete aspect *A2* which informs *Integer* as the actual type parameter to its super-aspect *A*. Thus, in place of *E*, the weaver uses *Integer* to check whether members have or not a match. In the presence of *A2*, method *m1* (declared at line 2 of Listing 2.16) is advised by the advice declared in line 5 of the same example.

2.3 Final Considerations

This chapter discussed the main design decisions of Generic Java and Generic AspectJ type systems. For sure that raw types and type erasure guarantee backward compatibility and simplify the migration of a non-generic code to take advantage of the generic support. But, the price of this compatibility is that a full and sound reification of the generic type system is not possible, at least while the migration is taking place (GOSLING et al., 2005).

Both raw types and type erasure cause the absence of type information during the execution of the program. The AspectJ language deals with this by disallowing the use of parameterized type in some constructs or simply warning the programmer during compilation.

According to Dincklage and Diwan (2004), Munsil (2004) and Donovan *et al.* (2004), the task related to adding actual type parameter to generic classes declarations and instantiations is complex and error prone. When aspects are considered,

[11] In the lightweight calculus proposed by Jagadeesan et al. (2006) (named as Aspect Featherweight Generic Java (AFGJ)) the weaving of generic aspects is allowed.

subtleties involving both the absence of type information, inter-type declarations, advices and pointcuts must be taken into account since they may affect the result of the refactoring. Therefore, tools to help this migration are essential to minimize the transformation effort and to prevent the introduction of new errors.

The remainder of this dissertation focus on a proposal that gives the main steps to automatize this process. While Chapter 3 presents an aspect-aware type constraint framework – which generates constraints between types of expressions and declarations comprising each program construct –, Chapter 4 describes an iterative type inference algorithm which aims to provide automatically actual type parameter for each raw declaration.

3 TYPE CONSTRAINT RULES

Type constraints are a formalism for expressing subtype relationship between program entities. This formalism can be used for type checking, type inference, preserving the type-correctness of programs in refactorings, and others. Consider the assignment $x = y$. In this example, the constraint $[y] \leq [x]$ states that the type of y (represented by $[y]$) must be a subtype of the type of x. Hence, the assignment $x = y$ is type-correct only if that constraint holds.

This chapter extends a model of type constraints used and modified by several authors (TIP et al., 2003; FUHRER et al., 2005; KIEŻUN et al., 2007; SUTTER; DOLBY, 2004); which was initially presented by Palsberg and Schwartzbach (1993). This extension proposes a set of constraint generation rules for the polymorphic version of AspectJ. These rules are concerned with deriving types and imposing constraints between the types of expressions and the types of declarations.

The main foundation of this work (FUHRER et al., 2005) focuses on OO code, more specifically on Java language. Deriving type constraints for AspectJ is significantly more complex than studies concentrated on pure Java programs: beyond all OO subtleties, it involves the use of aspects to encapsulate crosscutting concerns and working with a vague semantics of advice matching[1]. This requires non-trivial modification of the original type constraint rules, including most notably:

- the matching simulation for capturing advices being combined with the code;
- the analysis of AspectJ's parent declarations in subtype relationships;
- the analysis of AspectJ's inter-type declarations in almost all program constructs;
- the introduction of wildcard constraint variables to accommodate wildcard types.

In the remainder of this chapter, P denotes the original program and type constraints are generated from its abstract syntax tree (AST). For each program construct in P a non-empty set of constraints that expresses relationship between types is generated. These constraints expose the restrictions that P must have to be type-correct. By definition, a program is type-correct if the type constraints of all constructions are satisfied (FUHRER et al., 2005).

[1] It is not clear, even for the AspectJ development team, how matching should work in the presence of generic types. See the AspectJ bug `https://bugs.eclipse.org/bugs/show_bug.cgi?id=253109`.

3.1 Basic Concepts and Functions

Before the framework of type constraints be presented, it is important to describe the notation, basic functions and concepts required to the best comprehension of the rules discussed in this chapter.

v	variables	T	type variables
m	method names	I	interfaces
f	field names	A	aspects
M	methods	AD	advices
F	fields	E	expressions/declarations
C	classes, interfaces or aspect	τ, α	types

Figure 3.1: Notation

The main notation used in this work is depicted in Figure 3.1. Let M and F be a method and a field declaration, respectively. These declarations include the complete signature and the reference to declaring class (or aspect). If they represent inter-type declarations, the *on type* is also available. Let C represent both generic and non-generic classes, interfaces or aspects, depending on each context. Let T represent type variables, A aspects, AD advices and E expressions or declarations. Finally, consider τ and α as any valid AspectJ type (i.e. classes, interfaces, aspects, type variables and wildcards).

$[E]$	type of expression/declaration E	Decl(F)	type which declares field F
$[E]_p$	type of E in the original program	Param(M,i)	the i^{th} formal parameter of method M
$[M]$	declared type of method return	Param(M,e)	the formal parameter of method M which is related to args(..e..)
$[F]$	declared type of field	Param(AD,e)	the formal parameter of advice AD which is related to args(..e..)
T(E)	actual type for T in expression E	NrParams(M)	number of parameters of method M
T(C)	actual type for T in class C		
Decl(M)	type which declares method M		

Figure 3.2: Basic functions

Figure 3.2 depicts a set of basic functions for obtaining information about: the type of expressions ($[E]$ and $[E]_p$); the declared type of a field or the return of a method ($[F]$ and $[M]$, respectively); the actual type parameter bound to a given type variable T ($T(E)$ and $T(C)$); the type declaring a method or a field ($Decl(M)$ and $Decl(F)$); and the formal parameter declared in methods or advices ($Param(M,i)$, $Param(M,e)$ and $Param(AD,e)$).

$$
bounds(\tau, rec) = \begin{cases}
\{Object\} & \text{if } \tau \text{ is declared as } T \\
\{bounds(\tau_0, rec), .., bounds(\tau_n, rec)\} & \text{if } \tau \text{ is declared as T extends } \tau_0 \text{ \&,..,\& } \tau_n \ \wedge \ rec = true \\
\{\tau_0, .., \tau_n\} & \text{if } \tau \text{ is declared as T extends } \tau_0 \text{ \&,..,\& } \tau_n \ \wedge \ rec = false \\
\{C\} & \text{if } \tau \text{ is } C \\
\{C\langle \tau_1, .., \tau_n \rangle\} & \text{if } \tau \text{ is } C\langle \tau_1, .., \tau_n \rangle
\end{cases}
$$

The function presented above is used to obtain all existing upper bounds of a given type variable. If the first argument, τ, is not a type variable (i.e. it is a reifiable, a cooked or a raw type) the function returns the parameter itself. Otherwise, if τ is a type variable, the function returns the set of bounds defined in τ's declaration. Note that this function accepts a second argument called *rec*. When *rec* is *true*, *bounds*(..) is executed recursively up until a concrete type for each bound of τ be found.

$$bound(\tau) \quad = first(bounds(\tau, true))$$

Analogous to *bounds*(..), the above function is also used to obtain the upper bound of a type variable. The main difference between them is that *bound*(..) is concerned with retrieving only the first bound, while *bounds*(..) retrieves the whole set of bounds. In current Java specification (GOSLING et al., 2005), the first concrete bound is the one put in place of the corresponding type variable during the erasure process.

$$wildbound(\tau) = \begin{cases} \tau' & if\ \tau\ is\ ?\ extends\ \tau' \\ \tau' & if\ \tau\ is\ ?\ super\ \tau' \\ \tau & otherwise \end{cases}$$

There are some constraint generation rules that will need to retrieve the bound of a given wildcard. This task is performed by the above *wildbound*(..) function. The main goal of this function is to retrieve the upper bound when τ is an *extends* wildcard and the lower bound when τ is a *super* wildcard.

$$\begin{array}{ll} head(C\langle \tau_0, .., \tau_n \rangle) & = C \\ head(C) & = C \end{array}$$

The *head*(..) function is the one responsible for returning the raw type of a given cooked type. Note that since only classes, aspects and interfaces are valid parameters, it is undefined for type variables. Hence, before calling the *head*(..) function with a type variable it is necessary to get the bound of such variable as it is shown in the following definition.

$$|\tau| \quad = head(bound(\tau))$$

The above function simulates the GJ type erasure process by removing all actual type parameters from parameterized types. In summary, it returns the raw version of τ's most important bound (i.e. the first concrete bound). Consider the following as some examples of the applicability of this function.

Examples of erasure function usage

$|List\langle Integer\rangle|$ = *List*
$|T|$ `where` T `is declared as:` T *extends MyClass & MyInterface* = *MyClass*
$|Integer|$ = *Integer*

The erasure of the parameterized type $List\langle Integer\rangle$, is its raw *List*. The erasure of the type variable T is its first concrete bound, *MyClass*. The erasure of the reifiable type *Integer* is itself.

$$OnType(M) = \begin{cases} \tau & if\ Decl(M)\ is\ an\ Aspect \quad \wedge \quad M\ is\ an\ inter-type\ decl\ \tau.m \\ Decl(M) & otherwise \end{cases}$$

$$OnType(F) = \begin{cases} \tau & if\ Decl(F)\ is\ an\ Aspect \quad \wedge \quad F\ is\ an\ inter-type\ decl\ \tau.f \\ Decl(F) & otherwise \end{cases}$$

The *OnType*(..) function depicted above is the responsible for discovering what is the declaration type of a method (or field) independently if it is an inter-type declaration or not. In other words, when a given method M (or a field F) is declared in an aspect A but on behalf of a class C, this function always returns C. For example:

Listing 3.1: Inter-type declaration

```
1  class MyClass{..}
2  aspect MyAspect{
3      void MyClass.m(){..}
4  }
```

Listing 3.1 shows method *m* being implicitly declared on *MyClass* through the inter-type declaration defined in line 3. Thus, the *OnType*(..) function returns *MyClass* when it is called passing *m* as argument. Note that the type declaring *m* is *MyAspect* but the *on type* is *MyClass*.

$$SuperClass(C) = \begin{cases} C_1 & \text{if } C \text{ extends } C_1 \\ C_2 & \text{if } \exists_1 \text{ Aspect } A \text{ that declares parent: } C \text{ extends } C_2 \\ A_1 & \text{if } C \text{ is an Aspect} \quad \wedge \quad C \text{ extends } A_1 \quad \wedge \quad A_1 \text{ is an abstract Aspect} \\ Object & \text{otherwise} \end{cases}$$

$$Interfaces(C) = \begin{cases} \{I \mid C \text{ implements } I\} \\ \quad \cup \{I_1 \mid \text{Aspect } A \text{ declares parent: } C \text{ implements } I_1\} & \text{if } C \text{ is a Class} \quad \vee \quad \text{an Aspect} \\ \{I_2 \mid C \text{ extends } I_2\} \\ \quad \cup \{I_3 \mid \text{Aspect } A \text{ declares parent: } C \text{ extends } I_3\} & \text{if } C \text{ is an Interface} \end{cases}$$

Similarly to *OnType*(..) function, *SuperClass*(..) and *Interfaces*(..) are responsible for discovering what are the supertypes of a given class, interface or aspect – independently if they are parent declarations or not. Note that these functions are only defined for classes, interfaces and aspects. In other words, both *SuperClass*(..) and *Interfaces*(..) are undefined for type variables because type variables do not have supertypes, they only have bounds. See the following listing as an example:

Listing 3.2: Parent declaration

```
1  class C{..}
2  interface I{..}
3  aspect A{
4      declare parents: C implements I;
5  }
```

Listing 3.2 shows the aspect *A* declaring the interface *I* on behalf of class *C* (line 4). Therefore, calling *SuperClass*(*C*) results in *Object* and calling *Interfaces*(*C*) results in the set $\{I\}$. Note that these results are only possible because both functions take into account the use of aspects to encapsulate crosscutting concerns.

When aspects are considered, the subtype relationship may be affected by parent declarations. In face of that, the rules that define such relation are shown as follow.

$$\tau_0 \leq \tau_0 \qquad (s1)$$

$$\tau_0 \leq \tau_n \quad iff \quad \tau_0 \leq \tau_1 \quad \wedge \quad .. \quad \wedge \quad \tau_{n-1} \leq \tau_n \qquad (s2)$$

$$C_0 \leq C_1 \quad iff \quad C_1 = SuperClass(C_0) \quad \vee \quad C_1 \in Interfaces(C_0) \qquad (s3)$$

$$T_0 \leq \tau_1 \quad iff \quad \tau_1 \in bounds(T_0, false) \qquad (s4)$$

$$\tau_0\langle\alpha'_0,..,\alpha'_k\rangle \leq \tau_1\langle\alpha_0,..,\alpha_n\rangle \quad iff \quad \tau_0 \leq \tau_1 \quad \wedge \quad \tau_1 \text{ is declared as } C\langle T_0,..,T_n\rangle \quad \wedge \quad T_i(\tau_0) = T_i(\tau_1) \quad \wedge \quad 0 \leq i \leq n \qquad (s5)$$

$$\tau_0\langle\alpha_0,..,\alpha_k\rangle \leq \tau_1 \quad iff \quad \tau_0 \leq \tau_1 \quad \wedge \quad (\tau_1 \text{ is a raw type} \quad \vee \quad \tau_1 \text{ is not a generic type}) \qquad (s6)$$

$$\tau_0 \leq \tau_1\langle\alpha_0,..,\alpha_k\rangle \quad iff \quad \tau_1\langle\alpha_0,..,\alpha_k\rangle \leq \tau_0 \qquad (s7)$$

The subtyping relationship ($\leq$) is the reflexive and transitive closure ($s1$ and $s2$, respectively) of the *extends/implements* relation between types. Relation $s3$ states that a given type C_0 is a subtype of another type C_1 if C_0 *extends/implements* directly or indirectly – through a parent declaration – C_1. Relation $s4$ says that a given type variable T_0 is a subtype of any of its bounds. Relation $s5$ assumes that whenever

there is a type τ_0 that is a subtype of a generic type τ_1, all type parameters defined in τ_1 must be invariant. Relation S6 states that the cooked type $C\langle Integer\rangle$, for example, is a subtype of the raw type C. To provide compatibility with old code that instantiates classes without type parameters, raw types are used in place of cooked types even knowing that these constructions are not safe. The rule for this unsafe type judgments is provided by relation $s7$.

Since the above subtyping relationship understands parent declarations, it is possible to create a definition of AspectJ's method overriding as follows:

Definition 3.1.1 *A virtual method M_0 in type $\tau_0 \equiv OnType(M_0)$ overrides a virtual method M_1 in type $\tau_1 \equiv OnType(M_1)$ iff M_0 and M_1 have identical signatures or M_0 is a sub-signature*[2] *of M_1 and $\tau_0 \leq \tau_1$ but $\tau_0 \neq \tau_1$.*

The above definition allows the creation of the $RootDefs(..)$ function, which returns the roots of a method declaration, independently of whether the hierarchy is built by aspects using parent declaration or not.

$$RootDefs(M) = \{OnType(M_1) \mid M \text{ overrides } M_1 \quad \wedge \quad \text{there exists no } M_2\ (M_2 \neq M_1) \text{ such that } M_1 \text{ overrides } M_2\}$$

Finally, it is presented as follows the syntax of type constraints. Each side of the relationship is called *constraint variable*. A constraint variable is a type associated with a program construct which must be one of the following: (i) a type constant; (ii) the type of an expression; (iii) the declared type of a method or a field; or (iv) the formal parameter declared in a method.

$\tau_0 = \tau_1$ *type τ_0 must be the same as type τ_1*
$\tau_0 \leq \tau_1$ *type τ_0 must be the same as, or a subtype of type τ_1*

3.2 Aspect-Aware Type Constraints

This section presents a set of rules for deriving type constraints from Java and AspectJ programs. These rules extend Fuhrer *et al.*'s (2005) proposal by providing the generation of constraints for wildcards and the most important constructs of AspectJ. Additionally, these rules accommodate subclasses of generic classes and generate constraints for OO constructions that were not considered in the original proposal: raw types used in the signature of methods declared into generic classes and calls to methods defined in generic superclasses.

The strategy used in the remainder of this chapter for the generation of type constraints is based on small steps. This means that, instead of checking if a generic type C_1 is a subtype of another generic type C_2, the verification is broken into little pieces. As an example, consider the assignment shown in line 5 of Listing 3.3.

Listing 3.3: Generic types may be complex

```
1  class C1<F1,F2> {..}
2  class C2<F3,F4> extends C1<F3,F4>{..}
3  ..
4  C2<Integer, C2<Integer, Number>> y = null;
5  C1<Integer, ? extends C1<Integer, ? super Double>> x = y;
```

[2]The notion of sub-signature is used to express a relationship between two methods whose signatures are not identical, but in which one overrides the other (GOSLING et al., 2005).

As it was seen previously, the constraint $[y] \leq [x]$ must be satisfied in order to an assignment be type-correct . However, the generic types used in the above example make the solution of this simple constraint somewhat difficult. Therefore, in order to identify if $[y]$ is a subtype of $[x]$, it is necessary to compare each type parameter individually as it can be seen in Figure 3.3.

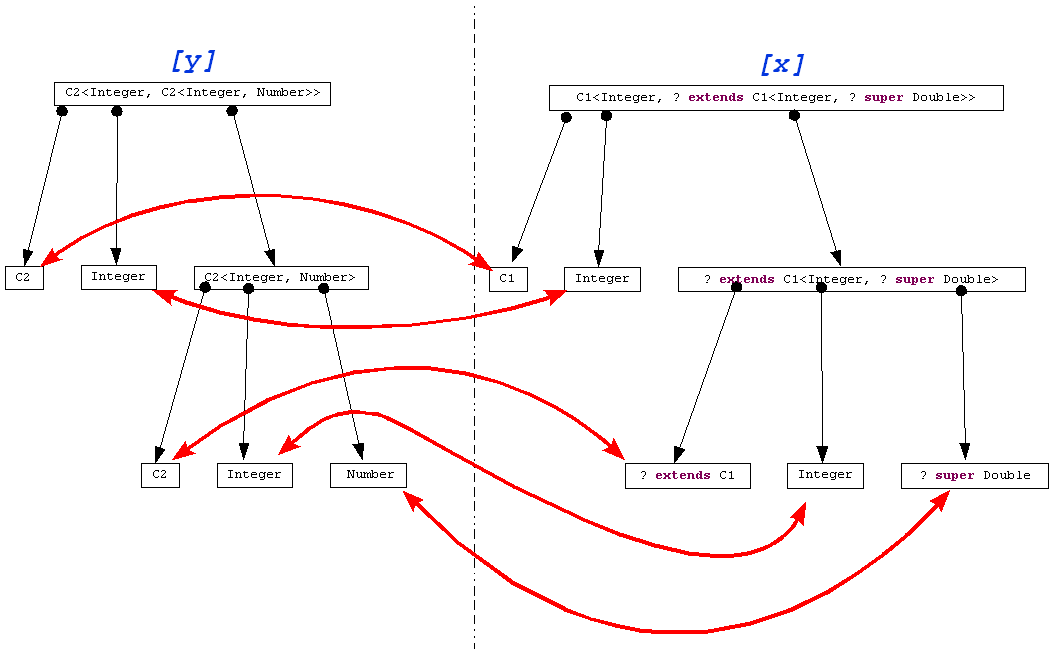

Figure 3.3: Constraints between generic types

This figure shows how the type of y (left hand side) must be constrained with the type of x (right hand side). To simplify the resolution, the same strategy is used in this dissertation. Thus, instead of generating a unique restriction saying that $[y]$ must be a subtype of $[x]$, this work generates constraints for each part of each type. In other words, each red arrow represents a type constraint that must be satisfied.

3.2.1 Assignment, Cast and Expression Rules

For creating constraints between expressions and declarations the *NonContextCgen*(..) function presented below is used in a recursive manner. This function is defined by case analysis on its third argument, τ. Cases $n1-n3$ are applied when τ is a reifiable type (e.g. *Integer*, *Double*, etc). Cases $n4-n6$ are applied when τ is a type parameter. Cases $n7$ and $n8$ are applied when τ is an upper- or a lower-bounded wildcard. Note that in $n7$ and $n8$ the function makes a recursive call changing the fourth argument, *useWild*, to indicate that wildcard type constraints must also be generated. Finally, $n9-n11$ are applied when τ is a parameterized type and $n12-n14$ when τ is a raw type.

$$
\begin{array}{l}
NonContextCgen(\alpha, op, \tau, useWild) = \\
\left\{\begin{array}{llll}
\alpha \ op \ C & & when\ \tau \equiv C \ \wedge \ useWild = \bullet & (n1) \\
\alpha \ op \ C & \vee \ \alpha \in \{?\ extends\ \tau' \mid \tau' \leq C\} & when\ \tau \equiv C \ \wedge \ useWild = \triangleleft & (n2) \\
\alpha \ op \ C & \vee \ \alpha \in \{?\ super\ \tau' \mid \tau' \geq C\} & when\ \tau \equiv C \ \wedge \ useWild = \triangleright & (n3) \\
 & & & \\
\alpha = T & & when\ \tau \equiv T \ \wedge \ useWild = \bullet & (n4) \\
\alpha = T & \vee \ \alpha \in \{?\ extends\ T' \mid T' \leq T\} & when\ \tau \equiv T \ \wedge \ useWild = \triangleleft & (n5) \\
\alpha = T & \vee \ \alpha \in \{?\ super\ T' \mid T' \geq T\} & when\ \tau \equiv T \ \wedge \ useWild = \triangleright & (n6)
\end{array}\right.
\end{array}
$$

Continue...

$$
\begin{array}{l}
NonContextCgen(\alpha, op, \tau, useWild) = \\
\left\{\begin{array}{lll}
NonContextCgen(\alpha, \leq, \tau', \triangleleft) & when\ \tau \equiv ?\ extends\ \tau' & (n7) \\
NonContextCgen(\alpha, \geq, \tau', \triangleright) & when\ \tau \equiv ?\ super\ \tau' & (n8) \\
 & & \\
\alpha\ op\ C & when\ \tau \equiv C\langle \tau_1, .., \tau_k \rangle \ \wedge\ useWild = \bullet & (n9) \\
\quad \wedge\ NonContextCgen(T_i(\alpha), =, \tau_i, \bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k & \\
(\alpha\ op\ C \ \vee\ \alpha \in \{?\ extends\ head(C') \mid C' \leq C\}) & when\ \tau \equiv C\langle \tau_1, .., \tau_k \rangle \ \wedge\ useWild = \triangleleft & (n10) \\
\quad \wedge\ NonContextCgen(T_i(\alpha), =, \tau_i, \bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k & \\
(\alpha\ op\ C \ \vee\ \alpha \in \{?\ super\ head(C') \mid C' \geq C\}) & when\ \tau \equiv C\langle \tau_1, .., \tau_k \rangle \ \wedge\ useWild = \triangleright & (n11) \\
\quad \wedge\ NonContextCgen(T_i(\alpha), =, \tau_i, \bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k & \\
 & & \\
\alpha\ op\ C & when\ \tau \equiv rawtype\ C \ \wedge\ useWild = \bullet & (n12) \\
\quad \wedge\ (T_i(\alpha) = T_i(C) \ \vee\ T_i(C) \in Wild(T_i(\alpha))) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k & \\
(\alpha\ op\ C \ \vee\ (\alpha \in \{?\ extends\ head(C') \mid C' \leq C\}) & when\ \tau \equiv rawtype\ C \ \wedge\ useWild = \triangleleft & (n13) \\
\quad \wedge\ (T_i(\alpha) = T_i(C) \ \vee\ T_i(C) \in Wild(T_i(\alpha))) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k & \\
(\alpha\ op\ C \ \vee\ (\alpha \in \{?\ super\ head(C') \mid C' \geq C\}) & when\ \tau \equiv rawtype\ C \ \wedge\ useWild = \triangleright & (n14) \\
\quad \wedge\ (T_i(\alpha) = T_i(C) \ \vee\ T_i(C) \in Wild(T_i(\alpha))) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k &
\end{array}\right.
\end{array}
$$

It is important to say that constraints generated by this function respect the Java invariant subtyping. When variance is allowed (i.e. in declaration sites), this function generates wildcards accordingly. Note that in cases $n12-n14$ wildcard type constraints are generated using a recursive helper function called $Wild(..)$. The main goal of this function is to produce a set containing all possible wildcards based on a given type. Since $Wild(..)$ depends on $T_i(\alpha)$ (see $n12-n14$), this function must be executed only when a fully defined type be estimated for α – which normally happens after all constraint are generated in a constraint solver stage.

Listing 3.4: Simple generic hierarchy

```
1  class C1<F2> {}
2  class C2<F1> extends C1<F1>{}
```

Listing 3.4 shows a simple generic hierarchy in which $C2$ is a subtype of $C1$. Based on this hierarchy, two examples of $Wild(..)$ usage are presented below:

Wildcard generation examples

```
Example 1:
Wild(C2) = {? extends C2, ? extends C1, ? extends Object}

Example 2:
Wild(C2<C2>) = { C2<? extends C2>, C2<? extends C1>, C2<? extends Object>,
    ? extends C2<C2>, ? extends C2<? extends C2>, ? extends C2<? extends C1>,
    ? extends C2<? extends Object>, ? extends C1<C2>, ? extends C1<? extends C2>,
    ? extends C1<? extends C1>, ? extends C1<? extends Object>, ? extends Object}
```

In *Example 1* the set of wildcards is based on type $C2$, while in *Example 2* the set of wildcards is based on type $C2\langle C2\rangle$. Note that all possibilities of upper-bounded wildcards were generated by $Wild(..)$ function and that the order of this set – which is the lowest in the hierarchy must come first – is important because it simplifies the constraint resolution further discussed in Section 4.2.2.

While $NonContextCgen(..)$ accepts, in its second argument op, an operation that indicates what will be the relationship ($\leq$ for subtype and $=$ for equals) of the constraints, $RNonContextCgen(..)$ (depicted below) generates only type constraints that obligates α to be exactly equals to τ. Note that the recursion is very similar to $NonContextCgen(..)$ function with a subtle difference: it only generates equals relationship between types.

$$
\begin{array}{l}
RNonContextCgen(\alpha,\tau,useWild) = \\
\left\{\begin{array}{lll}
\alpha = C & when\ \tau \equiv C \ \wedge\ useWild = \bullet & (rn1)\\
\alpha = ?\ extends\ C & when\ \tau \equiv C \ \wedge\ useWild = \triangleleft & (rn2)\\
\alpha = ?\ super\ C & when\ \tau \equiv C \ \wedge\ useWild = \triangleright & (rn3)\\
 & & \\
\alpha = T & when\ \tau \equiv T \ \wedge\ useWild = \bullet & (rn4)\\
\alpha = ?\ extends\ T & when\ \tau \equiv T \ \wedge\ useWild = \triangleleft & (rn5)\\
\alpha = ?\ super\ T & when\ \tau \equiv T \ \wedge\ useWild = \triangleright & (rn6)\\
 & & \\
RNonContextCgen(\alpha,\tau',\triangleleft) & when\ \tau \equiv ?\ extends\ \tau' & (rn7)\\
RNonContextCgen(\alpha,\tau',\triangleright) & when\ \tau \equiv ?\ super\ \tau' & (rn8)\\
 & & \\
\alpha = C \ \wedge\ RNonContextCgen(T_i(\alpha),\tau_i,\bullet) & when\ \tau \equiv C\langle\tau_1,..,\tau_k\rangle \ \wedge\ useWild = \bullet & (rn9)\\
 & \quad where\ C\ is\ decl\ as\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\le}i{\le}k & \\
\alpha = ?\ extends\ C \ \wedge\ RNonContextCgen(T_i(\alpha),\tau_i,\bullet) & when\ \tau \equiv C\langle\tau_1,..,\tau_k\rangle \ \wedge\ useWild = \triangleleft & (rn10)\\
 & \quad where\ C\ is\ decl\ as\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\le}i{\le}k & \\
\alpha = ?\ super\ C \ \wedge\ RNonContextCgen(T_i(\alpha),\tau_i,\bullet) & when\ \tau \equiv C\langle\tau_1,..,\tau_k\rangle \ \wedge\ useWild = \triangleright & (rn11)\\
 & \quad where\ C\ is\ decl\ as\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\le}i{\le}k & \\
 & & \\
\alpha = C \ \wedge\ T_i(\alpha) = T_i(C) & when\ \tau \equiv rawtype\ C \ \wedge\ useWild = \bullet & (rn12)\\
 & \quad where\ C\ is\ decl\ as\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\le}i{\le}k & \\
\alpha = ?\ extends\ C \ \wedge\ T_i(\alpha) = T_i(C) & when\ \tau \equiv rawtype\ C \ \wedge\ useWild = \triangleleft & (rn13)\\
 & \quad where\ C\ is\ decl\ as\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\le}i{\le}k & \\
\alpha = ?\ super\ C \ \wedge\ T_i(\alpha) = T_i(C) & when\ \tau \equiv rawtype\ C \ \wedge\ useWild = \triangleright & (rn14)\\
 & \quad where\ C\ is\ decl\ as\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\le}i{\le}k &
\end{array}\right.
\end{array}
$$

Now, a few of the type constraint generation rules that use the functions discussed so far are presented as follow:

$$\frac{P\ contains\ explicit\ declaration\ of\ a\ variable:\ \tau\ \ v}{RNonContextCgen([v],\tau,\bullet)} \quad (\text{R1})$$

$$\frac{P\ contains\ cast\ expression\ E \equiv (C)E_0}{RNonContextCgen([E],C,\bullet)} \quad (\text{R2})$$

Rule R1 states that the type of a given variable v is the type of its declaration τ. For a casting expression $(C)E_0$, rule R2 says that the type of the entire expression $(C)E_0$ is the same as the target type C referred to in the cast. Note that both R1 and R2 use *RNonContextCgen*(..) to recursively generate type constraints because the program constructs related to these rules require, in order to be type-correct, that the types be exactly the same.

$$\frac{\begin{array}{c}P\ contains\ downcast\ expression\ E \equiv (C)E_0\\ C\ is\ not\ an\ interface \qquad [E_0]\ is\ not\ an\ interface\end{array}}{NonContextCgen([E_0],\geq,C,\bullet)} \quad (\text{R3})$$

$$\frac{P\ contains\ assignment\ E_1 = E_2}{NonContextCgen([E_2],\leq,[E_1],\bullet)} \quad (\text{R4})$$

Rule R3 is related to downcasts and it requires that C be a subtype of the type of the expression E_0 being casted. Rule R4 states that an assignment $E_1 = E_2$ is type-correct if the type of E_2 is a subtype of $[E_1]$. Note that both R3 and R4 allow subtype relationships. Hence, there is a set of valid types (instead of a unique type) that satisfies the constraints generated by these rules. Moreover, since R3 and R4 use *NonContextCgen*(..), the type being assigned (C for R3 and $[E_1]$ for R4) may also declare a wildcard.

Listing 3.5: Generic declaration and assignment

```
1  class C<T>{
2          void m(List<T> l){
```

```
3                   ArrayList<T> l2 = (ArrayList<T>)l;
4                   ...
5           }
6   }
```

Consider Listing 3.5 as a simple example to show how these few rules discussed up until now are used to generate type constraints. The piece of code presented in this listing has a generic variable declaration at line 2 ($List\langle T\rangle\ l$) and another at line 3 ($ArrayList\langle T\rangle\ l2$); and, a cast expression ($(ArrayList\langle T\rangle)\ l$) and an assignment ($l2 = (ArrayList\langle T\rangle)\ l$) also at line 3. First of all, take a look on both declarations.

Constraint generation for Listing 3.5 - declarations

```
Line 2 - declaration: List<T> l
  R1
  ⟶ RNonContextCgen([l], List<T>, •)
    ⟶rn9 [l] = List   ∧   RNonContextCgen(E([l]), T, •)
    ⟶rn4 [l] = List   ∧   E([l]) = T

Line 3 - declaration: ArrayList<T> l2
  R1
  ⟶ RNonContextCgen([l2], ArrayList<T>, •)
    ⟶rn9 [l2] = ArrayList   ∧   RNonContextCgen(E([l2]), T, •)
    ⟶rn4 [l2] = ArrayList   ∧   E([l2]) = T
```

When the AST is being visited, the generic declaration $List\langle T\rangle\ l$ is found. Then, applying R1 to this construct generates the following type constraints: $[l] = List \quad \wedge \quad E([l]) = T$. These constraints state that the type of variable l is $List$ and the actual type parameter E for variable l is T. Similarly, R1 is also applied to the generic declaration $ArrayList\langle T\rangle\ l2$ and the following constraints are generated: $[l2] = ArrayList \quad \wedge \quad E([l2]) = T$.

Constraint generation for Listing 3.5 - casting

```
Line 3 - casting: (ArrayList<T>) l
  R2
  ⟶ RNonContextCgen([(ArrayList<T>) l], ArrayList<T>, •)
    ⟶rn9 [(ArrayList<T>) l] = ArrayList   ∧   RNonContextCgen(E([(ArrayList<T>) l]),  T, •)
    ⟶rn4 [(ArrayList<T>) l] = ArrayList   ∧   E([(ArrayList<T>) l]) = T

  R3
  ⟶ NonContextCgen([l], ≥, ArrayList<T>, •)
    ⟶n9 [l] ≥ ArrayList   ∧   NonContextCgen(E([l]), =, T, •)
    ⟶n4 [l] ≥ ArrayList   ∧   E([l]) = T
```

For the casting expression $(ArrayList\langle T\rangle)\ l$, rules R2 and R3 are the responsible for generating type constraints. While R2 says that the whole cast expression has the type $ArrayList\langle T\rangle$ (see the application of $rn4$ in the above example), R3 says that the type of l must be a supertype of $ArrayList$ and the actual type parameter E for variable l must be equal to T (see the application of $n4$ in the same example).

Constraint generation for Listing 3.5 - assignment

```
Line 3 - assignment: l2 = (ArrayList<T>) l
  R4
  ⟶ NonContextCgen([(ArrayList<T>) l], ≤, [l2], •)
    ⟶NonContextCgen([(ArrayList<T>) l], ≤, ArrayList<T>, •)
    ⟶n9 [(ArrayList<T>) l] ≤ ArrayList   ∧   NonContextCgen(E([(ArrayList<T>) l]), =, T, •)
    ⟶n4 [(ArrayList<T>) l] ≤ ArrayList   ∧   E([(ArrayList<T>) l]) = T
```

In the above constraint generation, rule R4 is used to generate type constraints for the assignment $l2 = (ArrayList\langle T\rangle)\ l$. Since the type of $l2$ is already known (i.e. $ArrayList\langle T\rangle$), R4 generates a type constraint saying that the type of the cast expression must be a subtype of $ArrayList$ $([(ArrayList\langle T\rangle)\ l] \leq ArrayList)$ and the type parameter of the cast expression must be T $(E([(ArrayList\langle T\rangle)l]) = T)$.

Table 3.1: Summary of constraints generated for Listing 3.5

Code	*Type constraint(s)*	*Rule(s)*
$List\langle T\rangle\ l$	$[l] = List \quad \wedge \quad E([l]) = T$	R1
$ArrayList\langle T\rangle\ l2$	$[l2] = ArrayList \quad \wedge \quad E([l2]) = T$	R1
$(ArrayList\langle T\rangle)\ l$	$[(ArrayList\langle T\rangle)\ l] = ArrayList \quad \wedge \quad E([(ArrayList\langle T\rangle)\ l]) = T$	R2
	$ArrayList \leq [l] \quad \wedge \quad E([l]) = T$	R3
$l2 = (ArrayList\langle T\rangle)\ l$	$[(ArrayList\langle T\rangle)\ l] \leq ArrayList \quad \wedge \quad E([(ArrayList\langle T\rangle)\ l]) = T$	R4

Table 3.1 summarizes the full set of generics-related type constraints computed for the example shown in Listing 3.5. This table indicates what are the constraints that were generated for each program construct (in the first and the second columns) and what are the rules that were used in each case (in the last column).

Looking at this table it is possible to infer that the fragment of code presented in Listing 3.5 is type-correct since: according to R1, l has the type $List\langle T\rangle$ and $l2$ has the type $ArrayList\langle T\rangle$; according to R2, the cast expression $(ArrayList\langle T\rangle)\ l$ has the type $ArrayList\langle T\rangle$; Therefore, since $ArrayList$ is a subtype of $[l]$ (i.e. $List$) and $E([l])$ is equal to T, R3 holds; and since the type of casting expression is $ArrayList\langle T\rangle$, R4 also holds.

3.2.2 Method Call Rules

The constraint generation $ContextCgen(..)$ is the recursive helper function used to create type constraints for generic and non-generic method calls. It depends on the calling context (which is represented by the fourth argument E) to create constraints between the type of the method's declaration and the type of method's clients. Note that Furher *et al.* (2005) have defined a similar solution, but their proposal does not generate wildcard type constraints and it only works for generic classes[3]. The function presented below generates wildcard constraints when applicable and works for both generic and non-generic classes.

$$
\begin{array}{l}
ContextCgen(\alpha, op, \tau, E, useWild) = \\
\left\{\begin{array}{lll}
\alpha\ op\ C & when\ \tau \equiv C \ \wedge\ useWild = \bullet & (c1)\\
\alpha\ op\ C \ \vee\ \alpha \in \{?\ extends\ \tau' \mid \tau' \leq C\} & when\ \tau \equiv C \ \wedge\ useWild = \triangleleft & (c2)\\
\alpha\ op\ C \ \vee\ \alpha \in \{?\ super\ \tau' \mid \tau' \geq C\} & when\ \tau \equiv C \ \wedge\ useWild = \triangleright & (c3)\\
 & & \\
\alpha\ op\ T(E) & when\ \tau \equiv T \ \wedge\ useWild = \bullet & (c4)\\
\alpha\ op\ T(E) \ \vee\ \alpha \in \{?\ extends\ \tau' \mid \tau' \leq T(E)\} & when\ \tau \equiv T \ \wedge\ useWild = \triangleleft & (c5)\\
\alpha\ op\ T(E) \ \vee\ \alpha \in \{?\ super\ \tau' \mid \tau' \geq T(E)\} & when\ \tau \equiv T \ \wedge\ useWild = \triangleright & (c6)\\
 & & \\
ContextCgen(\alpha, \leq, \tau', E, \triangleleft) & when\ \tau \equiv ?\ extends\ \tau' & (c7)\\
ContextCgen(\alpha, \geq, \tau', E, \triangleright) & when\ \tau \equiv ?\ super\ \tau' & (c8)\\
 & & \\
\alpha\ op\ C & when\ \tau \equiv C\langle \tau_1, .., \tau_k\rangle \ \wedge\ useWild = \bullet & (c9)\\
\quad \wedge\ ContextCgen(T_i(\alpha), =, \tau_i, E, \bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1, ..., T_k\rangle \ \wedge\ 1 \leq i \leq k & \\
(\alpha\ op\ C \ \vee\ \alpha \in \{?\ extends\ head(C') \mid C' \leq C\}) & when\ \tau \equiv C\langle \tau_1, .., \tau_k\rangle \ \wedge\ useWild = \triangleleft & (c10)\\
\quad \wedge\ ContextCgen(T_i(\alpha), =, \tau_i, E, \bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1, ..., T_k\rangle \ \wedge\ 1 \leq i \leq k & \\
(\alpha\ op\ C \ \vee\ \alpha \in \{?\ super\ head(C') \mid C' \geq C\}) & when\ \tau \equiv C\langle \tau_1, .., \tau_k\rangle \ \wedge\ useWild = \triangleright & (c11)\\
\quad \wedge\ ContextCgen(T_i(\alpha), =, \tau_i, E, \bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1, ..., T_k\rangle \ \wedge\ 1 \leq i \leq k &
\end{array}\right.
\end{array}
$$

[3]Furher *et al.*'s (2005) proposal defines a set of rules for non-generic classes, another set of rules for generic classes and some closure rules to fill the gap between these sets.

Continue...

$$
\begin{array}{l}
\mathit{ContextCgen}(\alpha, op, \tau, E, \mathit{useWild}) = \\
\left\{\begin{array}{lll}
\alpha\ op\ C & & \mathit{when}\ \tau \equiv \mathit{rawtype}\ C \ \wedge\ \mathit{useWild} = \bullet \quad (c12) \\
\quad \wedge\ (T_i(\alpha) = T_i(C) \ \vee\ T_i(C) \in \mathit{Wild}(T_i(\alpha))) & & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\leq}i{\leq}k \\
(\alpha\ op\ C \ \vee\ (\alpha \in \{?\ \mathit{extends}\ \mathit{head}(C') \mid C' \leq C\}) & & \mathit{when}\ \tau \equiv \mathit{rawtype}\ C \ \wedge\ \mathit{useWild} = \triangleleft \quad (c13) \\
\quad \wedge\ (T_i(\alpha) = T_i(C) \ \vee\ T_i(C) \in \mathit{Wild}(T_i(\alpha))) & & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\leq}i{\leq}k \\
(\alpha\ op\ C \ \vee\ (\alpha \in \{?\ \mathit{super}\ \mathit{head}(C') \mid C' \geq C\}) & & \mathit{when}\ \tau \equiv \mathit{rawtype}\ C \ \wedge\ \mathit{useWild} = \triangleright \quad (c14) \\
\quad \wedge\ (T_i(\alpha) = T_i(C) \ \vee\ T_i(C) \in \mathit{Wild}(T_i(\alpha))) & & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\leq}i{\leq}k
\end{array}\right.
\end{array}
$$

Similarly to *NonContextCgen*(..), the above function generates type constraints based on its third argument, τ, for: non-generic and generic classes ($c1-c3$ and $c9-c14$, respectively); type parameters ($c4-c6$) and; wildcards ($c7$ and $c8$). The main difference between these two functions is centered in cases $c4$, $c5$ and $c6$. While *NonContextCgen*(..) works without any context, *ContextCgen*(..) uses its forth argument E to discover what are the actual type parameters being used by the method caller.

$$
\begin{array}{l}
\mathit{RContextCgen}(\alpha, \tau, E, \mathit{useWild}) = \\
\left\{\begin{array}{lll}
\alpha\ =\ C & & \mathit{when}\ \tau \equiv C \ \wedge\ \mathit{useWild} = \bullet|\circ \quad (rc1) \\
\alpha\ =\ ?\ \mathit{extends}\ C & & \mathit{when}\ \tau \equiv C \ \wedge\ \mathit{useWild} = \triangleleft \quad (rc2) \\
\alpha\ =\ ?\ \mathit{super}\ C & & \mathit{when}\ \tau \equiv C \ \wedge\ \mathit{useWild} = \triangleright \quad (rc3) \\
 & & \\
\alpha\ =\ T(E) & & \mathit{when}\ \tau \equiv T \ \wedge\ \mathit{useWild} = \bullet \quad (rc4) \\
\alpha\ =\ \mathit{wildbound}(T(E)) & & \mathit{when}\ \tau \equiv T \ \wedge\ \mathit{useWild} = \circ \quad (rc5) \\
\alpha\ =\ ?\ \mathit{extends}\ T(E) & & \mathit{when}\ \tau \equiv T \ \wedge\ \mathit{useWild} = \triangleleft \quad (rc6) \\
\alpha\ =\ ?\ \mathit{super}\ T(E) & & \mathit{when}\ \tau \equiv T \ \wedge\ \mathit{useWild} = \triangleright \quad (rc7) \\
 & & \\
\mathit{RContextCgen}(\alpha, \tau', E, \triangleleft) & & \mathit{when}\ \tau \equiv ?\ \mathit{extends}\ \tau' \quad (rc8) \\
\mathit{RContextCgen}(\alpha, \tau', E, \triangleright) & & \mathit{when}\ \tau \equiv ?\ \mathit{super}\ \tau' \quad (rc9) \\
 & & \\
\alpha\ =\ C \ \wedge\ \mathit{RContextCgen}(T_i(\alpha), \tau_i, E, \bullet) & & \mathit{when}\ \tau \equiv C\langle\tau_1,..,\tau_k\rangle \ \wedge\ \mathit{useWild} = \bullet|\circ \quad (rc10) \\
 & & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\leq}i{\leq}k \\
\alpha\ =\ ?\ \mathit{extends}\ C \ \wedge\ \mathit{RContextCgen}(T_i(\alpha), \tau_i, E, \bullet) & & \mathit{when}\ \tau \equiv C\langle\tau_1,..,\tau_k\rangle \ \wedge\ \mathit{useWild} = \triangleleft \quad (rc11) \\
 & & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\leq}i{\leq}k \\
\alpha\ =\ ?\ \mathit{super}\ C \ \wedge\ \mathit{RContextCgen}(T_i(\alpha), \tau_i, E, \bullet) & & \mathit{when}\ \tau \equiv C\langle\tau_1,..,\tau_k\rangle \ \wedge\ \mathit{useWild} = \triangleright \quad (rc12) \\
 & & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\leq}i{\leq}k \\
 & & \\
\alpha\ =\ C \ \wedge\ T_i(\alpha) = T_i(C) & & \mathit{when}\ \tau \equiv \mathit{rawtype}\ C \ \wedge\ \mathit{useWild} = \bullet|\circ \quad (rc13) \\
 & & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\leq}i{\leq}k \\
\alpha\ =\ ?\ \mathit{extends}\ C \ \wedge\ T_i(\alpha) = T_i(C) & & \mathit{when}\ \tau \equiv \mathit{rawtype}\ C \ \wedge\ \mathit{useWild} = \triangleleft \quad (rc14) \\
 & & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\leq}i{\leq}k \\
\alpha\ =\ ?\ \mathit{super}\ C \ \wedge\ T_i(\alpha) = T_i(C) & & \mathit{when}\ \tau \equiv \mathit{rawtype}\ C \ \wedge\ \mathit{useWild} = \triangleright \quad (rc15) \\
 & & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ C\langle T_1,..,T_k\rangle \ \wedge\ 1{\leq}i{\leq}k
\end{array}\right.
\end{array}
$$

Analogous to *RNonContextCgen*(..), the above function generates constraints that obligate the argument α to be exactly equal to τ. Note that this function differs from *RNonContextCgen*(..) when type parameters are considered (cases $rc4-rc7$). Before creating the type constraints, it uses E to discover what are the actual type parameters being passed by the method caller. Beyond such detail, the first recursion (which is represented by the symbol $\circ$ in the fourth argument *useWild*) is also differentiated because the type parameter defined in context E may be, in some cases, a wildcard. In those cases, getting information from a generic container (such as $\mathit{List}\langle ?\ \mathit{extends}\ \mathit{Number}\rangle$) must return the bound of the wildcard. Such condition will be better explained forward with the example of Listing 3.6.

Once these recursive helper functions were defined, the following rules are required to create type constraints for method constructs:

$$
\frac{\begin{array}{c}\tau = \mathit{OnType}(M) \\ M\ \mathit{contains\ an\ expression}\ E \equiv \mathit{this}\end{array}}{[E] = \tau} \qquad (\text{R5})
$$

Rule R5 states that the type of the expression *this* depends on M's *on type*. In other words, if there exists an inter-type declaration *void C.m*(){ *this.m*2(); } into an

aspect A, for example, the type of *this* is C even knowing that it was declared inside of A.

$$\frac{M \text{ contains an expression } return\ E_0}{NonContextCgen([E_0], \leq, [M], \bullet)} \quad (R6)$$

Since the use of wildcards is allowed in method declarations, rule R6 states that the return of the method M must be a wildcard or the type of the returning expression, $[E_0]$, must be a subtype of the method's return, $[M]$.

$$\frac{P \text{ contains call } E_0.m(E_1,..,E_k) \text{ to virtual method } M \quad RootDefs(M) = \{\tau_1,..,\tau_q\}}{[E_0] \leq \tau_1 \quad \vee \quad .. \quad \vee \quad [E_0] \leq \tau_q} \quad (R7)$$

$$\frac{1 \leq i \leq k \quad P \text{ contains call } E \equiv E_{rec}.m(E_1,..,E_k) \text{ to virtual method } M}{RContextCgen([E],[M]_p,E_{rec},\circ) \qquad ContextCgen([E_i],\leq,[Param(M,i)]_p,E_{rec},\bullet)} \quad (R8,R9)$$

$$\frac{1 \leq i \leq k \quad M' \text{ contains call } E \equiv m(E_1,..,E_k) \text{ to method } M \quad \alpha' \equiv OnType(M') \quad \alpha \equiv OnType(M) \quad \alpha' = \alpha}{RNonContextCgen([E],[M]_p,\bullet) \qquad NonContextCgen([E_i],\leq,[Param(M,i)]_p,\bullet)} \quad (R10,R11)$$

The above rules (R7-R11) are concerned with calls to methods in both generic and non-generic classes. For each method call, these rules create a set of constraints related to the method's return and to the method's parameters. While rules R7, R8 and R9 are responsible for creating constraints for virtual method calls, rules R10 and R11 are responsible for creating constraints for local method calls.

Rule R7 states that a declaration of a method with the same signature as M must occur in some supertype of $[E_0]$. The complexity in this rule stems from the fact that M may override one or more methods declared in supertypes $\tau_1 \ldots \tau_q$, and the type-correctness of the method call requires that the type of receiver expression E_0 be a subtype of one of these τ_i. This is expressed by way of a disjunction using the auxiliary function $RootDefs(..)$.

Rule R8 defines the type of the entire call-expression E to be the same as $M's$ return and R9 requires that the type of each parameter E_i be the same as or a subtype of the correspondent formal declaration $[Param(M,i)]_p$. Note that R8 uses $RContextCgen(..)$ and R9 uses $ContextCGen(..)$. This is required because both the return and the parameters of the method's declaration may define wildcards. As an example, take the following listing:

Listing 3.6: Method calls

```
1  class C{
2      void m(List<? extends Number> l){
3          Number n = l.get(0);
4          ..
5      }
6      public void main(String args[]){
7          List<Integer> lInt = ..
8          new C().m(lInt);
9
10         List<Double> lDouble = ..
11         new C().m(lDouble);
12     }
13 }
```

Note that Listing 3.6 depicts one call to method m passing a list of integers (line 8) and another call to the same method passing a list of doubles (line 11). In this

example, there are two interesting segments that must be highlighted. The first one is the variance of the parameter *List*⟨? *extends Number*⟩ on method *m* declaration (line 2) and the second one is the expression *l.get*(0) that is returning *Number* instead of ? *extends Number* (line 3). In order to demonstrate how the rules presented so far can be employed to generate constraints for these cases, take the following:

Constraint generation for Listing 3.6

```
Line 2 - declaration: List<? extends Number> l
  R1
  ⟶ RNonContextCgen([l], List<? extends Number>, •)
    rn9⟶ [l] = List   ∧   RNonContextCgen(E([l]), ? extends Number, •)
    rn7⟶ [l] = List   ∧   RNonContextCgen(E([l]), Number, ◁)
    rn2⟶ [l] = List   ∧   E([l]) = ? extends Number

Line 3 - declaration: Number n
  R1
  ⟶ RNonContextCgen([n], Number, •)
    rn1⟶ [n] = Number

Line 6 - declaration: List<Integer> lInt
  R1
  ⟶ RNonContextCgen([lInt], List<Integer>, •)
    rn9⟶ [lInt] = List   ∧   RNonContextCgen(E([lInt]), Integer, •)
    rn1⟶ [lInt] = List   ∧   E([lInt]) = Integer

Line 9 - declaration: List<Double> lDouble
  R1
  ⟶ RNonContextCgen([lDouble], List<Double>, •)
    rn9⟶ [lDouble] = List   ∧   RNonContextCgen(E([lDouble]), Double, •)
    rn1⟶ [lDouble] = List   ∧   E([lDouble]) = Double

Line 3 - virtual method call: l.get(0)
  R7
  ⟶ [l] ≤ List
  R8
  ⟶ RContextCgen([l.get(0)], E, l, ○)
    rc5⟶ [l.get(0)] = wildbound(E(l))
  R9
  ⟶ ContextCgen([0], ≤, int, l, •)
    c1⟶ [0] ≤ int
    ⟶ Integer ≤ int

Line 3 - assignment: n = l.get(0)
  R4
  ⟶ NonContextCgen([l.get(0)], ≤, [n], •)
    ⟶NonContextCgen([l.get(0)], ≤, Number, •)
    n1⟶ [l.get(0)] ≤ Number
```

The first four reductions show how R1 is used to create type constraints for variable declarations of Listing 3.6. Looking at these cases it is possible to see that: the type of the variable *n* is *Number*; the type of variables *l*, *lInt* and *lDouble* is *List* and; the actual type parameter *E* of *l*, *lInt* and *lDouble* are ? *extends Number*, *Integer* and *Double*, respectively;

The remainder of the reductions are related to line 3 of the same listing. Note that, in this line, the result of *l.get*(0) expression is being assigned to a local variable *n* – which type is *Number*. In order to that piece of code be type-correct, the constraint $[l.get(0)] \leq Number$ (created by rule R4) must be satisfied. The interesting point here is: the method *get*(..) declared into *List* returns *E* and the actual type parameter *E* defined for the variable *l* is, in this example, ? *extends Number*. Thus, the type of the entire expression *l.get*(0) is ? *extends Number*. Even knowing that there is no subtype relationship between ? *extends Number* and *Number* (see the subtyping definition on Section 3.1), the assignment $n = l.get(0)$ is considered well-typed. This

happens because R8 has generated the following constraint: $[l.get(0)] = wildbound(E(l))$. Since $E(l)$ is ? *extends Number* and $wildbound$(? *extends Number*) is *Number*, the constraint created by R4 ($[l.get(0)] \leq Number$) holds because the subtype relationship is reflexive. In other words, *Number* is a subtype of itself.

Another important item to discuss in this example is related to boxing and unboxing type conversions (GOSLING et al., 2005). In Java, every primitive type has an immutable wrapper object version. The conversion from primitive to wrapper is known as boxing and from wrapper to primitive is known as unboxing. In short, from a programming point of view, a primitive and its corresponding wrapper may be faced as the same type since the JVM is responsible to perform such conversion. That is the reason by which the constraint generated by rule R9 (i.e. $Integer \leq int$) also holds.

Besides these constraints, it is still necessary to create type constraints for calls to method *m* – which happens at lines 7 and 10:

Constraint generation for Listing 3.6 - virtual method calls

```
Line 7 - virtual method call: new C().m(lInt)
  R7
 ⟶ [new C()] ≤ C
  R9
 ⟶ ContextCgen([lInt], ≤, List<? extends Number>, new C(), •)
    c9
   ⟶ [lInt] ≤ List   ∧   ContextCgen(E([lInt]), =, ? extends Number, new C(), •)
       c7
      ⟶ [lInt] ≤ List   ∧   ContextCgen(E([lInt]), ≤, Number, new C(), ◁)
       c2
      ⟶ [lInt] ≤ List   ∧
                (E([lInt]) ≤ Number   ∨   E([lInt])  ∈  {? extends τ | τ ≤ Number})

Line 10 - virtual method call: new C().m(lDouble)
  R7
 ⟶ [new C()] ≤ C
  R9
 ⟶ ContextCgen([lDouble], ≤, List<? extends Number>, new C(), •)
    c9
   ⟶ [lDouble] ≤ List   ∧   ContextCgen(E([lDouble]), =, ? extends Number, new C(), •)
       c7
      ⟶ [lDouble] ≤ List   ∧   ContextCgen(E([lDouble]), ≤, Number, new C(), ◁)
       c2
      ⟶ [lDouble] ≤ List   ∧
                (E([lDouble]) ≤ Number   ∨   E([lDouble])  ∈  {? extends τ | τ ≤ Number})
```

In the above constraint generation, rule R7 states that the type of the constructor call *new* $C()$ must be less or equal to C. For expression *new* $C().m(lInt)$, rule R9 requires that the type of the variable *lInt* be a subtype of *List* and, the actual type parameter E be a subtype of *Number* or a wildcard. The same happens for the variable *lDouble* in expression *new* $C().m(lDouble)$.

$$\frac{\begin{array}{c}1 \leq i \leq k \\ P \text{ contains constructor call} \\ E \equiv new\ C(E_1,..,E_k) \text{ to constructor } M\end{array}}{RNonContextCgen([E],C,\bullet) \qquad ContextCgen([E_i],\leq,[Param(M,i)]_p,E,\bullet)} \quad \text{(R12, R13)}$$

Other important rules are R12 and R13. These rules are concerned to create constraints for constructor calls. Similar to method calls, rule R13 states that the type of each actual parameter E_i must be the same as or a subtype of the type of the corresponding formal parameter $[Param(M,i)]_p$. On the other hand, rule R12 says that the type of a constructor call *new* $C(E_1,..,E_k)$ has the same type defined for C. Note that *RNonContextCgen*(..) was used in R12. It is required because C may also be a generic type.

Constraint generation for Listing 3.6 - constructor call

```
Type of constructor call: new C()
  R12
  ⟶ RNonContextCgen([new C()], C, •)
    rn1
    ⟶ [new C()] = C
```

The above reduction shows how the type of a constructor call is derived. Note that Listing 3.6 contains two calls to method *m* (i.e. *new C().m(lInt)* and *new C().m(lDouble)*). In both cases, it is necessary to obtain the type of *new C()* in order to be able to conclude the constraints derived so far. For that, rule R12 must be applied and, as the result, it says that the type of *new C()* is equals to *C*. Note that, since *C* is a non-generic class, no constraints for type parameters were created.

Table 3.2 aims to summarize the result of the constraint generation for the piece of code depicted in Listing 3.6.

Table 3.2: Summary of constraints generated for Listing 3.6

Code	*Type constraint(s)*	*Rule(s)*
$List\langle ?\ extends\ Number\rangle\ l$	$[l] = List \ \wedge \ E([l]) = ?\ extends\ Number$	R1
$Number\ n$	$[n] = Number$	R1
$List\langle Integer\rangle\ lInt$	$[lInt] = List \ \wedge \ E([lInt]) = Integer$	R1
$List\langle Double\rangle\ lDouble$	$[lDouble] = List \ \wedge \ E([lDouble]) = Double$	R1
$l.get(0)$	$[l] \leq List$	R7
	$[l.get(0)] = Number$	R8
	$Integer \leq int$	R9
$n = l.get(0)$	$[l.get(0)] \leq Number$	R4
$new\ C().m(lInt)$	$[new\ C()] \leq C$	R7
	$[lInt] \leq List \ \wedge \ (E([lInt]) \leq Number \ \vee$	R9
	$E([lInt]) \in \{?\ extends\ \tau \mid \tau \leq Number\})$	
$new\ C().m(lDouble)$	$[new\ C()] \leq C$	R7
	$[lDouble] \leq List \ \wedge \ (E([lDouble]) \leq Number \ \vee$	R9
	$E([lDouble]) \in \{?\ extends\ \tau \mid \tau \leq Number\})$	
$new\ C()$	$[new\ C()] = C$	R12

Looking at this table it is possible to identify that $E(lInt)$ may be *Integer* (at row 3) or any subtype of *Number* (at row 7). Since the intersection between *Integer* and any subtype of *Number* is *Integer*, this type is assumed for $E(lInt)$. A similar situation happens for $E([lDouble])$, in which *Double* is the type selected. Therefore, since $E([lInt])$ is *Integer*, $E([lDouble])$ is *Double*, $[new\ C()]$ is *C* and *Integer* is the wrapper of *int*, it is possible to infer that all type constraints depicted in Table 3.2 are satisfied. Meaning that the fragment of code presented in Listing 3.6 is type-correct.

3.2.2.1 Static and Generic Methods

Constraint generation for static and generic method calls are very similar to those presented so far for "normal method"[4] calls. The main different between them are:

- *Generic method calls*: while rules for non-generic method calls lookup type variables from the class declaration, a generic method call must lookup those variables from the method declaration before going to the class.

[4]The term "normal methods" refer to non-generic and non-static methods.

- *Static method calls*: since static methods can not use type variables defined in the class, those variables must be looked exclusively up from the method's declaration.

It is important to highlight that the rules for both static and generic method calls differ from "normal method" ones only in the way they lookup the declaration of a given type variable reference. Since all other aspects remain the same this dissertation omits the detail of these rules for brevity.

3.2.3 Overriding Rules

Both Java and AspectJ languages support overriding of methods as described in Definition 3.1.1. Therefore, constraint generation rules for method overriding are a requisite to verify the type-correctness of classes and aspects. But, before going into details about these rules, it is necessary to show how actual type parameters are propagated in the GA hierarchy.

$$\frac{\begin{array}{c} T_i(\alpha)\ exists \qquad \alpha\ is\ declared\ as\ C\langle T_0,..,T_k\rangle \qquad 0 \le i \le k \\ C_2\langle \tau_0,..,\tau_n\rangle = SuperClass(C) \quad \vee \quad C_2\langle \tau_0,..,\tau_n\rangle \in Interfaces(C) \\ C_2\ is\ declared\ as\ C_2\langle T'_0,..,T'_n\rangle \qquad 0 \le j \le n \end{array}}{ContextCgen(T'_j(\alpha), =, \tau_j, \alpha, \bullet)} \tag{R14}$$

Rule R14 is concerned with subtype relationships among generic libraries where actual type parameters are propagated from the subtype to its supertype. This rule states that: if $T_i(\alpha)$ exists, all subtypes of α must provide the actual type information for T_i during the instantiation. Note that R14 uses function *ContextCgen*(..) to create such constraints. This is required because τ_j may be a complex generic type, and then, it must be treated in the same way as discussed in previous rules.

For overriding relationships, rules R15, R16 and R17 are used independently if the supertype is generic or not. These rules generate additional type constraints for overriding method's return (rule R15) and parameters (rule R16).

$$\frac{\begin{array}{c} M'\ overrides\ M \\ E'_i \equiv Param(M',i) \qquad E_i \equiv Param(M,i) \qquad 1 \le i \le NrParams(M') \\ \alpha' \equiv OnType(M') \qquad \alpha \equiv OnType(M) \end{array}}{\begin{array}{c} ContextCgen([M']_p, \le, [M]_p, \alpha', \bullet) \qquad RContextCgen([E'_i]_p, [E_i]_p, \alpha', \bullet) \\ If\ T(\alpha')\ exists,\ then\ T(\alpha')\ can\ not\ be\ a\ wildcard \end{array}} \tag{R15,R16,R17}$$

It is important to highlight that rule R17 states that wildcards are not allowed in $T(\alpha')$. This is required because, in some cases, rules R15 and R16 may generate a wildcard as a valid type for $T(\alpha')$. Since $T(\alpha')$ represents the type being propagated to the superclass, only concrete actual types are valid constructions. For this reason, R17 is used as a closure rule to forbid wildcards in such cases.

Another important subtle is about *RContextCgen*(..) function usage in rule R16. For method overriding, the case *rc*5 of this function can not be executed because getting the bound of a wildcard is only useful for retrieving data from a generic container. Then, instead of creating a new helper function with only one little difference, *RContextCgen*(..) was defined in such a way that passing $\bullet$ (instead of $\circ$), in the last argument, *rc*5 will never be executed.

Take the following listing as a simple example of overriding relationship among two generic classes:

Listing 3.7: Method overiding

```
1  class C1<T1>{
```

```
2       void m(List<? extends T1> l){ .. }
3   }
4
5   class C2<T2> extends C1<T2>{
6       void m(List<? extends T2> l){ .. }
7   }
```

In this example, $C2$ explicitly overrides the method m defined in $C1$. It is important to emphasize that even when the hierarchy is defined by an aspect (through a parent declaration), the overriding rules defined in this section are able to identify such detail (based on Definition 3.1.1), and therefore, type constraints are generated as follow:

Constraint generation for Listing 3.7

```
Line 2 and 6 - method overriding: C2.m(..) overrides C1.m(..)
  R16
  ⟶ RContextCgen([List<? extends T2>], List<? extends T1>, C2, •)
    rc10
    ⟶ [List<? extends T2>] = List   ∧   RContextCgen(E([List<? extends T2>]), ? extends T1, C2,•)
    rc8
    ⟶ [List<? extends T2>] = List   ∧   RContextCgen(E([List<? extends T2>]), T1, C2, ◁)
    rc6
    ⟶ [List<? extends T2>] = List   ∧   E([List<? extends T2>]) = ? extends T1(C2)
  R17
  ⟶ if T(C2) exists, then T(C2) can not be a wildcard
```

The generation of constraints shown above are related to the code presented in Listing 3.7. The two reductions (R16 and R17) are deriving type constraints for the overriding method m. Note that overriding method's parameters must be invariant. In other words, the type of a given parameter declared in an overriding method must be exactly the same of the one declared in the corresponding super definition. This is why rule R16 uses $RContextCgen(..)$ to generate constraints between the parameters of $C2.m$ and $C1.m$. The complexity of this rule stems from the fact that type variables and wildcards are allowed when generic classes are considered.

Note that R16 has generated a constraint saying that $E([List\langle ?\ extends\ T2\rangle])$ must be equal to $?\ extends\ T1(C2)$. Since $E([List\langle ?\ extends\ T2\rangle])$ is $?\ extends\ T2$, the constraint generated by R16 could be shown as: $?\ extends\ T2 = ?\ extends\ T1(C2)$. Based on this, it is simple to infer that $T2$ is equals to $T1(C2)$ (meaning that, the actual type parameter defined for $T1$ must be equals to the type variable $T2$). Since $C2$ is always passing $T2$ to its superclass $C1$ (see line 5 of Listing 3.7), the invariance is guaranteed and then R16 holds. Rule R17, though, is only necessary to forbid that wildcards (possibly generated by rule R16) be propagated from $C2$ to $C1$.

$$\frac{\begin{array}{c}1 \le i \le k \\ M' \text{ contains call } E \equiv m(E_1,..,E_k) \text{ to method } M \\ \alpha' \equiv OnType(M') \qquad \alpha \equiv OnType(M) \qquad \alpha' \le \alpha \qquad \alpha' \neq \alpha\end{array}}{\begin{array}{c}RContextCgen([E],[M]_p,\alpha',\circ) \qquad ContextCgen([E_i],\le,[Param(M,i)]_p,\alpha',\bullet) \\ If\ T(\alpha')\ exists,\ then\ T(\alpha')\ can\ not\ be\ a\ wildcard\end{array}} \quad (\text{R18,R19,R20})$$

Although rules R18, R19 and R20 are not directly related to overriding, they were designed in a very similar way than rules R15, R16 and R17. The biggest difference between them is that while rules R15-R17 address the constraint generation for overriding of methods, rules R18-R20 were created to manage calls of methods declared in supertypes.

3.2.4 Advices Rules

All constraint generation rules presented so far consciously manage inter-type and parent declarations. However, in order to provide an aspect-aware type constraint

framework, it is still important to define rules for advised methods. In AspectJ, an advice says what to do and a pointcut says when to do. These constructs, when used together, allow the definition of crosscutting concern to be specified in separate aspect modules.

Note that the weaving of advices takes place just after the type erasure process. Thus, at weaving-time, only *declared* generic type information are available to AspectJ. This restriction implies that only statically resolved constructions (i.e. *args*(..) pointcut primitives, $after()$ *returning*(..) advices and *around*(..) advices) can accept parameterized types. Since this dissertation focuses on the polymorphic version of AspectJ, designators which does not accept such types (i.e. *this*(..), *target*(..), $cflow$(..), etc) were not considered. Hence, a future study is required to make this type constraint framework to support all existing features of AspectJ.

The following rules were designed to generate type constraints for *args*(..) pointcut primitives (rule R21) and $after()$ *returning*(..) advices (rule R22). While rule R21 states that the parameters of an advised method[5] must be "castable" to the parameters defined in the *args* construct, rule R22 says that the return of an advised method must be "castable" to the parameter defined in the $after()$ *returing*(..) advice.

$$\frac{\begin{array}{c}\textit{method call/execution } M \textit{ is advised by } AD \\ AD\textit{'s pointcut expression contains } args(e_0,..,e_k) \\ E_i \equiv Param(M,e_i) \qquad E'_i \equiv Param(AD,e_i) \qquad 1 \leq i \leq n\end{array}}{ArgsCgen([E_i],[E'_i],\circ)} \tag{R21}$$

$$\frac{\begin{array}{c}\textit{method call/execution } M \textit{ is advised by } AD \\ AD \textit{ is an } after()\ \ returning(..) \textit{ advice} \qquad E \equiv Param(AD,0) \textit{ exists}\end{array}}{ArgsCgen([M],[E],\circ)} \tag{R22}$$

It is important to highlight that both rules use *ArgsCgen*(..) function to generate type constraints. Since AspectJ is not able to match against type variables, this function was designed to consider both AspectJ's exact matching (when types are fully defined in joinpoints) and AspectJ's erased matching (when type variables are used in joinpoints) as follows:

$$ArgsCgen(\alpha,\tau,useWild) = \begin{cases}
C \leq \tau \ \vee \ C >_? \tau & when\ \alpha \equiv C \ \wedge \ useWild = \circ & (ar1)\\
\tau = C \ \vee \ \tau \in Wild(C) & when\ \alpha \equiv C \ \wedge \ useWild = \bullet & (ar2)\\
\tau \in \{?\ extends\ C' \mid C \leq C'\} & when\ \alpha \equiv C \ \wedge \ useWild = \triangleleft & (ar3)\\
\tau \in \{?\ super\ C' \mid C \geq C'\} & when\ \alpha \equiv C \ \wedge \ useWild = \triangleright & (ar4)\\
 & & \\
|T| \leq \tau \ \vee \ |T| >_? \tau & when\ \alpha \equiv T \ \wedge \ useWild = \circ & (ar5)\\
\tau \in \{?\ extends\ \alpha' \mid |T| \leq \alpha'\} & when\ \alpha \equiv T \ \wedge \ useWild = \bullet & (ar6)\\
\tau \in \{?\ extends\ \alpha' \mid |T| \leq \alpha'\} & when\ \alpha \equiv T \ \wedge \ useWild = \triangleleft & (ar7)\\
\tau \in \{?\ super\ \alpha' \mid |T| \geq \alpha'\} & when\ \alpha \equiv T \ \wedge \ useWild = \triangleright & (ar8)\\
 & & \\
ArgsCgen(\alpha',\tau,\triangleleft) & when\ \alpha \equiv ?\ extends\ \alpha' & (ar9)\\
ArgsCgen(\alpha',\tau,\triangleright) & when\ \alpha \equiv ?\ super\ \alpha' & (ar10)\\
 & & \\
(C \leq \tau \ \vee \ C >_? \tau) & when\ \alpha \equiv C\langle\alpha_1,..,\alpha_k\rangle \ \wedge \ useWild = \circ & (ar11)\\
\quad \wedge \ ArgsCgen(\alpha_i,T_i(\tau),\bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1,..,T_k\rangle \ \wedge \ 1{\leq}i{\leq}k & \\
(\tau = C \ \vee \ \tau \in Wild(C)) & when\ \alpha \equiv C\langle\alpha_1,..,\alpha_k\rangle \ \wedge \ useWild = \bullet & (ar12)\\
\quad \wedge \ ArgsCgen(\alpha_i,T_i(\tau),\bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1,..,T_k\rangle \ \wedge \ 1{\leq}i{\leq}k & \\
\tau \in \{?\ extends\ head(C') \mid C \leq C'\} & when\ \alpha \equiv C\langle\alpha_1,..,\alpha_k\rangle \ \wedge \ useWild = \triangleleft & (ar13)\\
\quad \wedge \ ArgsCgen(\alpha_i,T_i(\tau),\bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1,..,T_k\rangle \ \wedge \ 1{\leq}i{\leq}k & \\
\tau \in \{?\ super\ head(C') \mid C \geq C'\} & when\ \alpha \equiv C\langle\alpha_1,..,\alpha_k\rangle \ \wedge \ useWild = \triangleright & (ar14)\\
\quad \wedge \ ArgsCgen(\alpha_i,T_i(\tau),\bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1,..,T_k\rangle \ \wedge \ 1{\leq}i{\leq}k &
\end{cases}$$

[5]This work assumes that the weaver is able to provide all matching information. For example, it can answer the following two questions: (i) what are the methods being advised by *AD*? and (ii) what are the advices advising *M*?

Continue...

$$
\begin{array}{l}
\mathit{ArgsCgen}(\alpha, \tau, \mathit{useWild}) = \\
\left\{\begin{array}{lll}
(C \leq \tau \ \vee \ C >_? \ \tau) & \mathit{when}\ \alpha \equiv \mathit{rawtype}\ C \ \wedge \ \mathit{useWild} = \circ & (ar15) \\
\quad \wedge \ (T_i(\tau) = T_i(C) \ \vee \ T_i(\tau) \in \mathit{Wild}(T_i(C))) & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ c\langle T_1,..,T_k\rangle \ \wedge \ 1{\leq}i{\leq}k & \\
(\tau \ = \ C \ \vee \ \tau \in \mathit{Wild}(C)) & \mathit{when}\ \alpha \equiv \mathit{rawtype}\ C \ \wedge \ \mathit{useWild} = \bullet & (ar16) \\
\quad \wedge \ (T_i(\tau) = T_i(C) \ \vee \ T_i(\tau) \in \mathit{Wild}(T_i(C))) & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ c\langle T_1,..,T_k\rangle \ \wedge \ 1{\leq}i{\leq}k & \\
\tau \in \{?\ \mathit{extends}\ \mathit{head}(C') \mid C \leq C'\} & \mathit{when}\ \alpha \equiv \mathit{rawtype}\ C \ \wedge \ \mathit{useWild} = \triangleleft & (ar17) \\
\quad \wedge \ (T_i(\tau) = T_i(C) \ \vee \ T_i(\tau) \in \mathit{Wild}(T_i(C))) & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ c\langle T_1,..,T_k\rangle \ \wedge \ 1{\leq}i{\leq}k & \\
\tau \in \{?\ \mathit{super}\ \mathit{head}(C') \mid C \geq C'\} & \mathit{when}\ \alpha \equiv \mathit{rawtype}\ C \ \wedge \ \mathit{useWild} = \triangleright & (ar18) \\
\quad \wedge \ (T_i(\tau) = T_i(C) \ \vee \ T_i(\tau) \in \mathit{Wild}(T_i(C))) & \quad \mathit{where}\ C\ \mathit{is\ decl\ as}\ c\langle T_1,..,T_k\rangle \ \wedge \ 1{\leq}i{\leq}k &
\end{array}\right.
\end{array}
$$

The above *ArgsCgen*(..) is very similar to those constraint generation helper functions discussed in previous sections. The main difference between them are: (i) the cases *ar*5-*ar*8, which simply perform the erasure of all type variables defined in the given joinpoint before creating any type constraint; and (ii) the first recursion (represented by $\mathit{useWild} = \circ$), which says that the type declared in the method being advised can be a subtype or a supertype of the type defined in the advice construct, with the price of a runtime verification for supertypes. Note that *ArgsCgen*(..) introduces this runtime verification by generating constraints using $>_?$ relation in cases *ar*1, *ar*5, *ar*11 and *ar*15.

Listing 3.8: Before advice using *args*(..)

```
1  class C<T extends Number>{
2      void m(List<? extends T> l){ .. }
3  }
4
5  aspect A{
6      before() : execution(void m(..)) &
7                 args(List<? extends Number>){
8          ..
9      }
10 }
```

In order to present an example of an advice matching against a generic class, Listing 3.8 shows the method *m* being advised by the advice declared at line 6. The interesting point of this example is that this advice is matching against a wildcard upper-bounded to a type variable. To generate type constraints for this piece of code, rule R21 must be applied as follows:

Constraint generation for Listing 3.8

```
Line 7 - args: args(List<? extends Number>)
  R21
  ⟶ ArgsCgen([List<? extends T>], [List<? extends Number>], ○)
    ar11⟶ (List ≤ [List<? extends Number>]   ∨   List >? [List<? extends Number>])   ∧
             ArgsCgen(? extends T, E([List<? extends Number>]), •)
    ar9⟶ (List ≤ [List<? extends Number>]   ∨   List >? [List<? extends Number>])   ∧
             ArgsCgen(T, E([List<? extends Number>]), ◁)
    ar7⟶ (List ≤ [List<? extends Number>]   ∨   List >? [List<? extends Number>])   ∧
             E([List<? extends Number>])  ∈  { ? extends α | |T| ≤ α  }
    ⟶ (List ≤ [List<? extends Number>]   ∨   List >? [List<? extends Number>])   ∧
             E([List<? extends Number>])  ∈  { ? extends α | Number ≤ α  }
    ⟶ (List ≤ [List<? extends Number>]   ∨   List >? [List<? extends Number>])   ∧
             E([List<? extends Number>])  ∈  { ? extends Number, ? extends Object}
```

In the first part of the conjunction of the resultant constraint it is possible to see that the type defined for the *args* argument must be a supertype or a subtype of *List*. Note that if it were a subtype of *List*, a runtime verification would be required. The second part of the conjunction is about its type parameter, which must be one

of the following: {? *extends Number*, ? *extends Object*}. Since the type of *args* is *List* and its type parameter is ? *extends Number*, all constraints are satisfied, and therefore, the code depicted in Listing 3.8 is type-correct.

One interesting and also questionable issue about AspectJ type system is related to the return of around advices. According to AspectJ Developers Notebook (2005), the return of an around advice must be assignable to the advised method's return. However, the compiler can not ensure type-correctness when type variables are used. The reason behind this statement is that, in current AspectJ implementation (version 1.6.3), advices can not use the same type variable reference used by the method. Therefore, since Java subtyping is invariant, there is no way to guarantee the type-correctness in such cases.

Listing 3.9: Around advice

```
1  class C<T extends Number> {
2          List<T> m1() {..}
3  }
4
5  aspect A{
6          List<Number> around(): execution(* C.m1()){
7                  System.out.println(``before method'');
8                  return proceed();
9          }
10 }
```

Listing 3.9 shows the method *m*1 being intercepted by an around advice. The body of this advice only prints a message on the default output and then calls the original method *m*1 through a *proceed* call (line 8). Note that, this advice is considered well-typed by AspectJ even knowing that $List\langle Number\rangle$ is not assignable to $List\langle T\rangle$. This happens because the advice's return is in fact assignable to an special erased version of the method's return – which is $List\langle Number\rangle$.

It is important to highlight that this strategy may cause type conversions errors at runtime. Suppose someone changes the line 8 to return a list containing *Double* instances and, in program *P*, there is a fragment of code as follows: *new* $C\langle Integer\rangle$.*m*1(*myIntList*). At compile-time it is type-correct but at runtime it will cause a type conversion error because *Double* is not assignable to *Integer*. The rule which reproduces this AspectJ typing strategy is R23[6]. This rule was created in such a way that constraints to enforce the advice's return to be a special erased version of the method's return be generated.

$$\frac{\textit{method call/execution } M \textit{ is advised by } AD \qquad AD \textit{ is an around advice}}{[AD] = Object \quad \vee \quad AroundCgen([AD], \leq, [M], \bullet)} \tag{R23}$$

In summary, this rule says that the return of an around advice must be *Object* or it must be assignable to the statically resolved method's return. A specific helper function, *AroundCgen*(..), was created to accomplish this task because every type variable must be replaced by its bound before type constraints be generated. Note that *AroundCgen*(..) makes basically the same that *NonContextCgen*(..) function does with one slight difference: it erasures type variables in cases *ad*4-*ad*6.

[6]Note that this work does not solve AspectJ typing issues, it only reproduces the existing behavior in the type constraint framework.

$$
\begin{array}{l}
AroundCgen(\alpha, op, \tau, useWild) = \\
\left\{\begin{array}{lll}
\alpha\ op\ C & when\ \tau \equiv C \ \wedge\ useWild = \bullet & (ad1)\\
\alpha\ op\ C \ \vee\ \alpha \in \{?\ extends\ \tau' \mid \tau' \leq C\} & when\ \tau \equiv C \ \wedge\ useWild = \triangleleft & (ad2)\\
\alpha\ op\ C \ \vee\ \alpha \in \{?\ super\ \tau' \mid \tau' \geq C\} & when\ \tau \equiv C \ \wedge\ useWild = \triangleright & (ad3)\\
 & & \\
\alpha\ =\ |T| & when\ \tau \equiv T \ \wedge\ useWild = \bullet & (ad4)\\
\alpha \leq |T| \ \vee\ \alpha \in \{?\ extends\ T' \mid T' \leq |T|\} & when\ \tau \equiv T \ \wedge\ useWild = \triangleleft & (ad5)\\
\alpha \geq |T| \ \vee\ \alpha \in \{?\ super\ T' \mid T' \geq |T|\} & when\ \tau \equiv T \ \wedge\ useWild = \triangleright & (ad6)\\
 & & \\
AroundCgen(\alpha, \leq, \tau', \triangleleft) & when\ \tau \equiv ?\ extends\ \tau' & (ad7)\\
AroundCgen(\alpha, \geq, \tau', \triangleright) & when\ \tau \equiv ?\ super\ \tau' & (ad8)\\
 & & \\
\alpha\ op\ C & when\ \tau \equiv C\langle \tau_1, .., \tau_k \rangle \ \wedge\ useWild = \bullet & (ad9)\\
\quad \wedge\ AroundCgen(T_i(\alpha), =, \tau_i, \bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k & \\
(\alpha\ op\ C \ \vee\ \alpha \in \{?\ extends\ head(C') \mid C' \leq C\}) & when\ \tau \equiv C\langle \tau_1, .., \tau_k \rangle \ \wedge\ useWild = \triangleleft & (ad10)\\
\quad \wedge\ AroundCgen(T_i(\alpha), =, \tau_i, \bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k & \\
(\alpha\ op\ C \ \vee\ \alpha \in \{?\ super\ head(C') \mid C' \geq C\}) & when\ \tau \equiv C\langle \tau_1, .., \tau_k \rangle \ \wedge\ useWild = \triangleright & (ad11)\\
\quad \wedge\ AroundCgen(T_i(\alpha), =, \tau_i, \bullet) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k & \\
 & & \\
\alpha\ op\ C & when\ \tau \equiv rawtype\ C \ \wedge\ useWild = \bullet & (ad12)\\
\quad \wedge\ (T_i(\alpha) = T_i(C) \ \vee\ T_i(C) \in Wild(T_i(\alpha))) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k & \\
(\alpha\ op\ C \ \vee\ (\alpha \in \{?\ extends\ head(C') \mid C' \leq C\}) & when\ \tau \equiv rawtype\ C \ \wedge\ useWild = \triangleleft & (ad13)\\
\quad \wedge\ (T_i(\alpha) = T_i(C) \ \vee\ T_i(C) \in Wild(T_i(\alpha))) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k & \\
(\alpha\ op\ C \ \vee\ (\alpha \in \{?\ super\ head(C') \mid C' \geq C\}) & when\ \tau \equiv rawtype\ C \ \wedge\ useWild = \triangleright & (ad14)\\
\quad \wedge\ (T_i(\alpha) = T_i(C) \ \vee\ T_i(C) \in Wild(T_i(\alpha))) & \quad where\ C\ is\ decl\ as\ C\langle T_1, .., T_k \rangle \ \wedge\ 1 \leq i \leq k &
\end{array}\right.
\end{array}
$$

The reduction presented below shows the constraint generation for the around advice of Listing 3.9. Note the disjunction of the resultant constraint: the type of the return of the advice must be *Object* or, it must be a subtype of *List* and the type parameter *E* must be equal to the erasure of the type variable *T*. Since the return type of the around advice (i.e. *List*⟨*Number*⟩) is a subtype of *List* and, *E*(*List*⟨*Number*⟩) is *Number* and |*T*| is also *Number*, the constraint generated by rule R23 holds.

Constraint generation for Listing 3.9 - around advice

```
Lines 2 and 6 - around a method: List<Number> around()
  R23
  ⟶ [List<Number>]=Object    ∨    AroundCgen([List<Number>], ≤, [List<T>], •)
    ad9
    ⟶ [List<Number>]=Object    ∨
                ([List<Number>] ≤ List    ∧    AroundCgen(E([List<Number>]), =, T, •))
    ad4
    ⟶ [List<Number>]=Object    ∨    ([List<Number>] ≤ List    ∧    E([List<Number>]) = |T|))
```

Another important construction is the implicit *proceed* method call. A *proceed*(..) call is allowed only into the body of around advices and, as the name suggests, the main idea of this construct is to proceed the original method execution. Thus, the constraint generation rule is defined as follows:

$$
\frac{\begin{array}{c}1 \leq i \leq k \\ AD\ contains\ a\ proceed\ call\ E \equiv proceed(E_1, .., E_k)\end{array}}{RNonContextCgen([E], [AD]_p, \bullet) \qquad NonContextCgen([E_i], \leq, [Param(AD, i)]_p, \bullet)} \quad (R24, R25)
$$

The signature of the *proceed*(..) method is exactly the same of the advice's signature. Hence, rules R24 and R25 generate constraints that are concerned with the return and with the parameters of a given *proceed*(..) call. Note that, instead of using context helper functions, these rules use non-context ones (i.e. *RNonContextCgen*(..) and *NonContextCgen*(..)). This is required because the type-correctness of a *proceed* call is verified against the advice's declaration, and therefore, no calling context exists.

Fraine *et al.* (2008) have found that this implicit *proceed* signature is the main responsible for undetected errors. Thus, they proposed the StrongAspectJ (an extension of AspectJ language with explicit *proceed* signature) to detect more type

errors at compile-time. Although StrongAspectJ is safer than AspectJ, this dissertation focuses on the current production ready solution – which is the version 1.6.3 of AspectJ language.

Constraint generation for Listing 3.9 - proceed

```
Line 8 - proceed call: proceed()
  R24
  ⟶ RNonContextCgen([proceed()], List<Number>, •)
    rn9
    ⟶ [proceed()] = List    ∧    RNonContextCgen(E([proceed()]), Number, •)
    rn1
    ⟶ [proceed()] = List    ∧    E([proceed()]) = Number

Line 8 - return: return proceed()
  R6
  ⟶ NonContextCgen([proceed()], ≤, List<Number>, •)
    n9
    ⟶ [proceed()] ≤ List    ∧    NonContextCgen(E([proceed()]), =, Number, •)
    n1
    ⟶ [proceed()] ≤ List    ∧    E([proceed()]) = Number
```

The reductions shown above depict the constraint generation for the *proceed* call of Listing 3.9. While the first rule (R24) generates type constraints for the *proceed*'s return, the second rule (R6) generates type constraints for the whole *return* expression. Finally, Table 3.3 summarizes the constraints generated for the piece of code of Listing 3.9.

Table 3.3: Summary of constraints generated for Listing 3.9

Code	*Type constraint(s)*	*Rule(s)*
$List\langle Number\rangle$ *around*()	$[List\langle Number\rangle] = Object \quad \vee$ $([List\langle Number\rangle] \leq List \quad \wedge \quad E([List\langle Number\rangle] = \lvert T\rvert))$	R25
proceed()	$[proceed()] = List \quad \wedge \quad E([proceed()]) = Number$	R24
return proceed()	$[proceed()] \leq List \quad \wedge \quad E([proceed()]) = Number$	R6

3.2.5 Other Rules

There are some type constraint rules that were not presented in this chapter. These rules are related to basic constructions such as: field, arrays, instanceof expressions and etc.

$$\frac{\begin{array}{c}\alpha = OnType(F) \\ P \text{ contains field access } E \equiv E_0.f \text{ to } F\end{array}}{RContextCgen([E], =, [F]_p, E_0, \circ) \qquad [E_0] \leq \alpha} \quad (\text{R26,R27})$$

For field access expressions ($E_0.f$) which access the field F – explicitly or implicitly (through an aspect) declared in type α –, rule R26 defines that the type of the whole expression must be the same as the one declared in F. Rule R27, though, requires that the type of expression E_0 be a subtype of field F's *on type*.

It is important to point out that these two rules (R26 and R27) are applied only for non-local field access. Likewise method calls, fields also require rules for local access and for super access. For brevity, these rules were not detailed in this dissertation, but they are very similar to those defined for methods (R10 and R18).

This chapter only summarizes the essential details of an aspect-aware type constraint framework and refer the reader to the foundations of this work (TIP et al., 2003; FUHRER et al., 2005) to analyze basic Java rules (such as, instanceof expression, hide fields, arrays and others). The omitted rules involve no significant

different analysis from the original proposal. However, it is important to emphasize that: (i) in order to consider inter-type declarations, the use of *Decl*(..) function must be replaced by the aspect-aware *OnType*(..) defined in Section 3.1; (ii) the subtyping relationship, presented in the same section, must also be applied because it takes into account declare parent constructs.

3.3 Final Considerations

This chapter presented an aspect-aware type constraint framework focused on the polymorphic version of AspectJ. The main foundation of this work (FUHRER et al., 2005) is concentrated on OO code, more specifically on Java programs. Unfortunately, unlike Java, AspectJ does not have a safe type system. As it was discussed in Section 3.2.4, both the return of advices and the implicit *proceed* signature can give rise to type errors at runtime. In addition, AspectJ's typing rules severely restrict advice definitions obligating the matching against type variables to happen against their erasure.

Because of these AspectJ issues and other subtleties related to aspect constructs and wildcards, the type constraint framework presented in this dissertation is significantly more complex than previous studies focused on pure OO programs. Therefore, a non-trivial modification of the original rules (FUHRER et al., 2005) was required to improve the existing support to generate constraints for the following:

- subclasses of generic classes[7];
- raw types used in signature of methods declared into generic classes;
- calls for methods defined in the super generic class;
- wildcard types;
- methods and attributes into aspects;
- inter-type declarations;
- declare parent declarations;
- advices constructs;
- and, *args*(..) primitive pointcut constructs.

It is important to highlight that during the conception of this dissertation, no other type constraint framework for the AspectJ language was found. Therefore, the solution discussed in this chapter is the first to propose a set of inference rules for generating type constraints from AspectJ programs.

In an effort to show a practical example of the presented aspect-aware solution, next chapter discusses how to extend an existing generic code migration to consider the use of aspects to encapsulate crosscutting concerns. It is important to say that the proposed solution is not restricted to this example. Hence, other works may also get benefits from this proposal since type constraints are being used to

[7] Although Fuhrer *et al.* (2005) claim being able to accommodate subclasses of generic classes, they do not show the type rules neither in the paper nor in the implementation.

solve many different problems in the literature. Some examples are: traditional inference algorithms (PALSBERG; SCHWARTZBACH, 1991; PLEVYAK; CHIEN, 1994; EIFRIG et al., 1995; AGESEN, 1995), which focus on inferring types for programs with no type annotations; refactoring algorithms that use pre- and post-conditions to guarantee that the program's behavior is preserved (TIP et al., 2003); algorithms for analyzing Java bytecode (BELLAMY et al., 2008); and others.

4 GENERIC CODE MIGRATION

The parametric polymorphism in both Java and AspectJ improves the type safety and the expressiveness of the source code. However, to legacy and non-generic code take advantage of this pervasive feature, they must be migrated to explicitly supply actual type parameters in both declarations and instantiations of generic classes.

Although the type systems of these languages were designed to support such migration, this process – when performed manually – can be tedious, time consuming and error prone (DINCKLAGE; DIWAN, 2004; MUNSIL, 2004; DONOVAN et al., 2004). The reason behind this assertion is that actual type parameters must be inferred to remove as much unsafe downcasts as possible without affecting the original semantics of the program. This is known in the generic migration literature as the *instantiation problem* (DONOVAN et al., 2004).

There are several approaches that aim to help solving this problem (DONOVAN et al., 2004; DINCKLAGE; DIWAN, 2004; FUHRER et al., 2005; CRACIUN et al., 2009). In general, they analyze the source code looking for gaps of type information and then rewrite a new semantically equivalent generic version of the program, in which actual type parameters are automatically inserted and redundant downcasts are removed.

These solutions have been successfully used for migrating pure object-oriented software, but they are not able to ensure that the migration will be successful in the presence of aspects. There are several subtleties involving both AspectJ's type system, inter-type declarations, parent declarations and pointcut expressions that make this task more complex and error prone. Then, approaches focused on the aspect-oriented context must deal with the following:

1. **Poor inference:** aspect-oriented constructs must be considered during the program transformation since they may change the inference results. Without analyzing such constructs, the inferred types may not be as good as expected and the code inside aspects will not be improved.

2. **Ill-typing:** since inter-type and parent declarations are implicitly woven into the application, actual type parameters inferred may become ill-typed when the weaver adapts the structure or the hierarchy of a class.

3. **Pointcut's fragility:** pointcuts can match elements across the entire base code. Thus, reasoning about the correctness of a pointcut requires a deep understanding of the internal structure of the software (YE; VOLDER, 2008; WLOKA et al., 2008). Therefore, the task of adding actual type parameters to raw declarations and instantiations, for example, can be troublesome.

4. **Inference of wildcards:** since aspects are specifically designed to deal with crosscutting concerns, the use of wildcards in some declarations (such as, *args*(..) pointcut primitive and *after*() *returning*(..) advices) must be preferred over specific types because they allow the removal of more insecure downcasts.

5. **An unsafe type system:** As it was discussed in the previous chapter, both the return of advices and the implicit *proceed* signature can give rise to type errors at runtime. In addition, AspectJ's typing rules severely restrict advice definitions obligating the matching against type variables to happens against their erasure.

Note that the generic code migration of AspectJ programs is a challenge because, besides difficulties related to Java, it must consider the above problems while simultaneously rewriting existing declarations and instantiations. In face of that, this chapter describes the use of the aspect-aware type constraint framework previously introduced in Chapter 3 along with the Fuhrer *et al.*'s (2005) algorithm to address the instantiation problem in the AO context, enabling then, the conversion of non-generic legacy code to add actual type parameters in both Java and AspectJ languages.

The following sections are organized as follows. Section 4.1 shows some motivational examples to explain each of the above problems. Section 4.2 presents an overview of the Fuhrer *et al.*'s (2005) algorithm and what is necessary to adapt in order to make it understand aspects and wildcards. Section 4.3 concludes the chapter with a discussion about the proposed aspect-aware generic code migration.

4.1 Motivational Examples

Before describing the migration algorithm, a couple of examples are shown to demonstrate some of the problems that must be taken into account during generic migration of AspectJ programs.

4.1.1 Poor Inference

In the aspect-oriented programming (AOP) paradigm, it is common to use aspects to deal with crosscutting concerns. Usually, these aspects use inter-type and parent declarations to introduce state or behavior to existing classes. Without analyzing such constructs during the generification[1], the inferred types may not be as good as expected (i.e. may not remove as must unsafe downcast as expected) and the code within aspects will not be improved to take advantage of a better compile-time type checking.

Listing 4.1: Poor inference

```
1 class C{
2     void m1(List l){ .. }
3     public static void main(String[] args) {
4         List l = new ArrayList();
5         l.add(1);
6         new C().m1(l);
7     }
8 }
```

[1]The term *generification* is used in this dissertation as the process related to convert a non-generic code into a new generic version.

```
 9  aspect A{
10      void C.m2(){
11          List l = new ArrayList();
12          l.add(1.1);
13          m1(l);
14      }
15  }
```

Listing 4.1 shows a simple example that contains two method declarations: (i) *m1*, which is declared into *C* at line 2; and (ii) *m2*, which is declared by aspect *A* on behalf of *C* through an inter-type declaration at line 10. Note that the *main* method is calling *m1* (line 6) passing a *List* containing an integer. Similarly, the method *m2* is also calling *m1* (line 13) but passing a *List* containing a double.

If aspect *A* (lines 9-15) is not considered during the generic migration of this code, the actual type parameter of the *List* declared in method *m1* (line 2) should be erroneously inferred as *Integer*. It may happens because the body of the inter-type declaration (lines 11-13) is not analyzed, and therefore, only one client of the method *m1* (which is passing a list of integers) is found.

It is important to highlight that, even though type *Integer* be inferred, the code will be considered type-correct. This occurs because no migration will be performed within aspect *A*, and therefore, the code in the body of method *m2* will be kept raw. As it was seen in Section 3.1, unsafe relationships between raw and parameterized types are allowed in both Java and AspectJ type systems[2].

4.1.2 Ill-typing

Although there are some AspectJ code that may be converted to start using generics by the cost of a poor inference, there are others that obligate AO constructs to be taken into account during the transformation. Without considering such constructs, the program will not be type-correct when the migration is finished. These cases are directly related to overriding and invariant subtyping. As an example of this problem, consider Listing 4.2.

Listing 4.2: Ill-typing

```
1  class C1 { .. }
2  class C2 extends C1 {
3      void m(List l){..}
4      public static void main(String[] args) {
5          List l = new ArrayList();
6          l.add(1);
7          new C2().m(l);
8      }
9  }
10 aspect A{
11     void C1.m(List l){..}
12 }
```

This listing shows the class *C2* extending *C1* (line 2) and declaring the method *m* (line 3). Moreover, the aspect *A* is declaring another method, also named as *m*, on behalf of *C1* through an inter-type declaration (line 11). After this aspect be woven into *C1*, the method *m* declared in *C2* will be overriding the method *m* implicitly declared in *C1*.

[2]When the compiler finds an unsafe construct it only warns the programmer saying that the statement is not type-safe.

It is important to say that all overriding relationships – including those defined by aspects – must be considered during a generic migration, otherwise the refactored version of the code will be ill-typed. In other words, without considering the inter-type declaration defined at line 11 there is no method overriding in Listing 4.2, and therefore, the migration will not take into account this important relationship. Thus, a tool (or a developer) may infer *Integer* as the actual type parameter for the *List* declared at line 3 because there is a call to method *m* passing a list of integers (line 7). It is important to note that such inference causes a type error because it breaks the notion of sub-signature required in overriding relationships (see Definition 3.1.1).

4.1.3 Pointcut's Fragility

Pointcut is a complex mechanism that, in general, is hard to write and maintain (YE; VOLDER, 2008; WLOKA et al., 2008). One reason for this assertion is that pointcuts may match elements across the entire base code, and therefore, reasoning about pointcut's correctness requires a global understanding of the internal structure of the software at a level of detail that is hard to obtain and difficult to remember. Another reason is that the AspectJ's pointcut language allows complex expression constructions and its semantics has several non-intuitive subtleties (YE; VOLDER, 2008; RUBBO et al., 2008).

During code transformation, pointcut's matching can be affected since any non-local change may modify the set of joinpoint being matched. This issue is widely discussed in the AO refactoring literature (MONTEIRO; FERNANDES, 2005, 2006; KOPPEN; STÖRZER, 2004; HANNEMANN et al., 2003; WLOKA et al., 2008) and, frequently, it is referred as the fragile pointcut problem. The generification is not an exception, hence the task of adding actual type parameters to aspect-oriented programs can be troublesome.

Listing 4.3: Pointcut's fragility

```
1  class C{
2      void m1(List l){..}
3      void m2(List l){..}
4  }
5  aspect A{
6      before(List l) : execution(void  C.*(..))
7          && args(l){ .. }
8  }
```

Listing 4.3 shows an example of the pointcut's fragility. In this example, the advice declared at line 6 is matching both methods *m1* and *m2* (lines 2 and 3, respectively). The task of adding a type parameter for the raw *List* declared in any of these methods must consider the type declared in the *args* construction. Otherwise, the semantics of the program can be modified without warnings. This can happens because that advice may stop matching one of the methods. Note that, after the generification all advices must be applying to exactly the same joinpoints they applied to before the refactoring had started.

4.1.4 Inference of Wildcards

Since wildcards provide a variance for the invariant Java subtyping – accepting a set of types instead of only a unique type –, the inference of wildcards in some declarations (such as, *args*(..) pointcut primitives and *after*() *returning*(..) advices) is

recommended to improve the type safety of aspect-oriented programs. This is based on the assumption that a unique pointcut can match many different joinpoints. Then, in these cases, the use of a place holder which allows a set of types is usually a better choice than a specific type.

Listing 4.4: Inference of wildcards

```
1  class C{
2      void m1(List l){..}
3      void m2(List l){..}
4
5      public static void main(String args[]){
6          List l1 = new ArrayList();
7          l1.add(1);
8          new C().m1(l1);
9
10         List l2 = new ArrayList();
11         l2.add(1.1);
12         new C().m2(l2);
13     }
14 }
15 aspect A{
16     before() : execution(void  C.m*(..))
17         && args(List){ .. }
18 }
```

In the above listing there are two method calls: the former is calling *m*1 passing a list of integers (line 8) and the latter is calling *m*2 passing a list of doubles (line 12). Moreover, it declares an advice (line 16) that will be triggered by the execution of any method declared in *C* whose name starts with `m` – in this example, methods *m*1 and *m*2. As it was seen in the previous section, the types inferred for methods being advised may affect the type chosen for the *args* pointcut primitive. Therefore, the *args* construct must be constrained with the whole collection of matched joinpoints. This implies that, if *List*⟨*Integer*⟩ were inferred for *m*1 and *List*⟨*Double*⟩ were inferred for *m*2, the best choice for *args*, in this example, would be *List*⟨? *extends Number*⟩.

Note that the wildcard inference assures that the type declared in each site are as tight as possible, in the sense that they are as low in the hierarchy as the type rules allow. If a non-wildcard strategy were chosen[3], the type inferred for the *args* primitive should be *List*⟨*Number*⟩. Knowing that the Java subtyping is invariant, this inference would obligate the *List* parameter of both methods *m*1 and *m*2 to also be declared as *List*⟨*Number*⟩. It is important to highlight that this invariant solution is quite poor because few insecure downcasts can be removed when more than one joinpoint is matched by a given advice.

4.1.5 Unsafe Type System

Unfortunately the AspectJ language does not have a safe type system (see Section 3.2.4). This means that current type rules are not able to ensure the type-correctness of a program when proceed calls or around advices matching against type variables are used.

Listing 4.5: Unsafe type system

```
1  class C<T extends Number> {
2      List<T> m( List<T> l) { ... }
3  }
```

[3]It is important to say that the main foundation of this dissertation (FUHRER et al., 2005) does not infer wildcards.

```
4  aspect A{
5      List around(): execution(List m(..)) {
6          List l = proceed();
7          l.add(1);
8          return l;
9      }
10 }
11 class Main {
12     public static void main(String args[]){
13         List l1 = new ArrayList();
14         l1.add(1.1);
15         List l2  = new C().m(l1);
16         for (Iterator iter=l2.iterator(); iter.hasNext();) {
17             Double d = (Double)iter.next();
18             ...
19         }
20     }
21 }
```

Listing 4.5 shows an example of the around advice issue, in which *C* is a generic class declaring a type variable *T*. This type variable is being used by method *m* in both parameter and return (line 2). When *m* is executed, the around advice declared at line 5 will be triggered, and then, an integer will be added to the end of the returning list.

Note that, in the *main* method, a list of doubles is being passed as an argument to *m* and the same type is expected to return (line 15). However, since the advice adds to the returning list an integer value, this code will raise a runtime exception when line 17 is reached.

Further, in Section 4.3.2, it will be shown that even though AspectJ considers the generic version of Listing 4.5 type-correct, it is still an unsafe construction that will cause a type conversion error at runtime. This happens because, in the current version of AspectJ (version 1.6.3) there is no way to check if around advices matching against type variables are type-correct.

4.2 The Algorithm

There are several proposals (DONOVAN et al., 2004; FUHRER et al., 2005; DINCKLAGE; DIWAN, 2004; TIP et al., 2004; CRACIUN et al., 2009) for migrating clients of generic classes to inform actual type parameters. Since these approaches focus on Java programs, they are not able to ensure that the migration will be successful in the presence of aspects. Therefore, this section proposes some modifications to Fuhrer *et al.*'s (2005) algorithm[4] in order to enable the conversion of non-generic legacy code to add actual type parameters in both Java and AspectJ programs.

The sequence diagram shown in Figure 4.1 presents the general idea of the Fuhrer *et al.*'s (2005) solution. When a developer (the *User* in the diagram) decides to perform a generic migration in his code, the method *refactor* of class *InferTypeArgumentsRefactoring* is called. This method abstracts the complexity behind the inference algorithm by receiving the compilation units[5] (CUs) selected to be refactored and by returning the new generic version of these resources. Other

[4]Fuhrer *et al.*'s (2005) work was chosen as the basis for this dissertation because it is the most practical solution in the sense that it uses an efficient algorithm and is implemented as a refactoring in Eclipse (www.eclipse.org).

[5]A compilation unit is the source code of a class or an aspect.

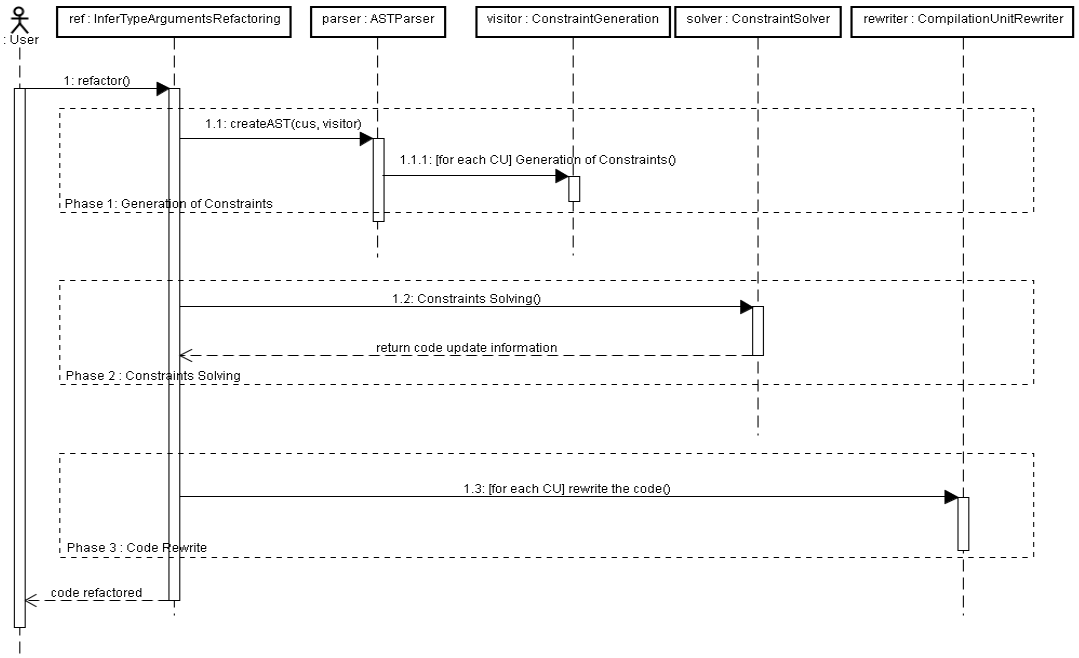

Figure 4.1: Algorithm overview

classes depicted in the diagram (i.e. *ASTParser*, *ConstraintGeneration*, *ConstraintSolver* and *CompilationUnitRewriter*) will be further discussed in next sections along with more details about the algorithm. The main idea of this figure is to show that the algorithm is divided in the following three phases:

1. *Generation of Constraints*: this very first phase is responsible for creating type constraints for each program construct;

2. *Constraints Solving*: is responsible for resolving all type constraints in order to determine: (i) a type for each declaration/instantiation and; (ii) what casts are redundant;

3. *Code Rewrite*: is responsible for rewriting the program's source code. It uses the information resulting from the previous phase to add type parameters to generic declarations and to remove redundant casts.

Each of these phases are detailed in the next sections. The main idea is to provide an overall understanding of each part of the algorithm and a discussion about what is necessary to modify in the original solution to make it start considering the use of aspects and wildcards. For that, Section 4.2.1 describes how constraints are generated. Section 4.2.2 presents how constraints generated in the first phase are resolved and Section 4.2.3 shows how the code is rewritten.

4.2.1 Generation of Constraints

The first part of the algorithm is the one responsible for generating type constraints for each program construct. These constraints express relationships between the types of expressions and declarations. In other words, these constraints make

a similar work that a compiler does to check if the program is type-correct. Figure 4.2 depicts, in a sequence diagram, the main steps of this phase.

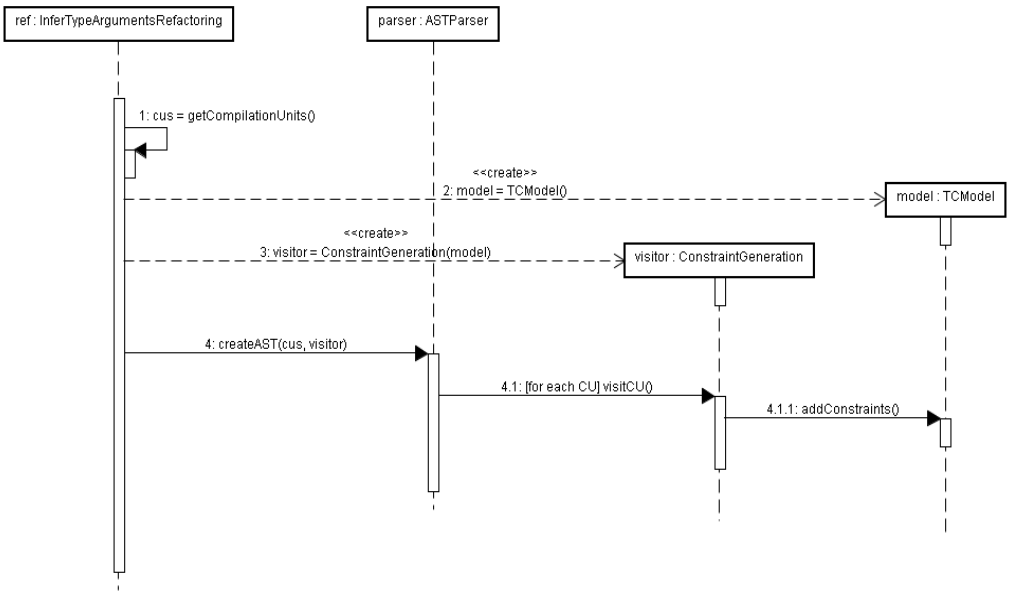

Figure 4.2: Algorithm overview - Phase 1

First of all, in step 1, the compilation units selected by the user to be the target of the refactoring are retrieved. Second, a *TCModel* is created to store information about type constraints, constraint variables and CUs. In the third step, the class *ConstraintGeneration* is instantiated. This class implements the Visitor design pattern (GAMMA et al., 1995) and defines a method for each program construct (such as assignment, method declaration, field access, etc) that is specialized to generate constraints for that piece of code. The fourth and last step creates an AST per CU. In this step, every node of the AST is visited and the corresponding method defined in *ConstraintGeneration* object is executed. Thus, type constraints are generated and stored into the *TCModel* object, which is the output of this phase.

The main difference between the original solution and the one proposed in this work is centered on the *ConstraintGeneration* class. While the original solution uses the rules presented in (FUHRER et al., 2005), the AO version must implement the constraint generation rules discussed in Chapter 3. Beyond that, the parse used to create an AST for each compilation unity must also be improved to support aspect modules.

4.2.2 Constraints Solving

Type constraints created in the first phase are iterated until a set of legal types for each constraint variable is computed. Figure 4.3 depicts a sequence diagram containing the overall idea of this phase.

Initially, in step 1, an object of the type *ConstraintSolver* is created. This object retrieves from the *TCModel* a collection with all constraint variables which comprise the type constraints generated in the previous phase.

In step 2, the method *solveConstraints* is called. First, an initial type estimate is associated with each constraint variable (step 2.1), which may be one of the following: (i) a singleton set containing a specific type (for type declaration, constants, literals

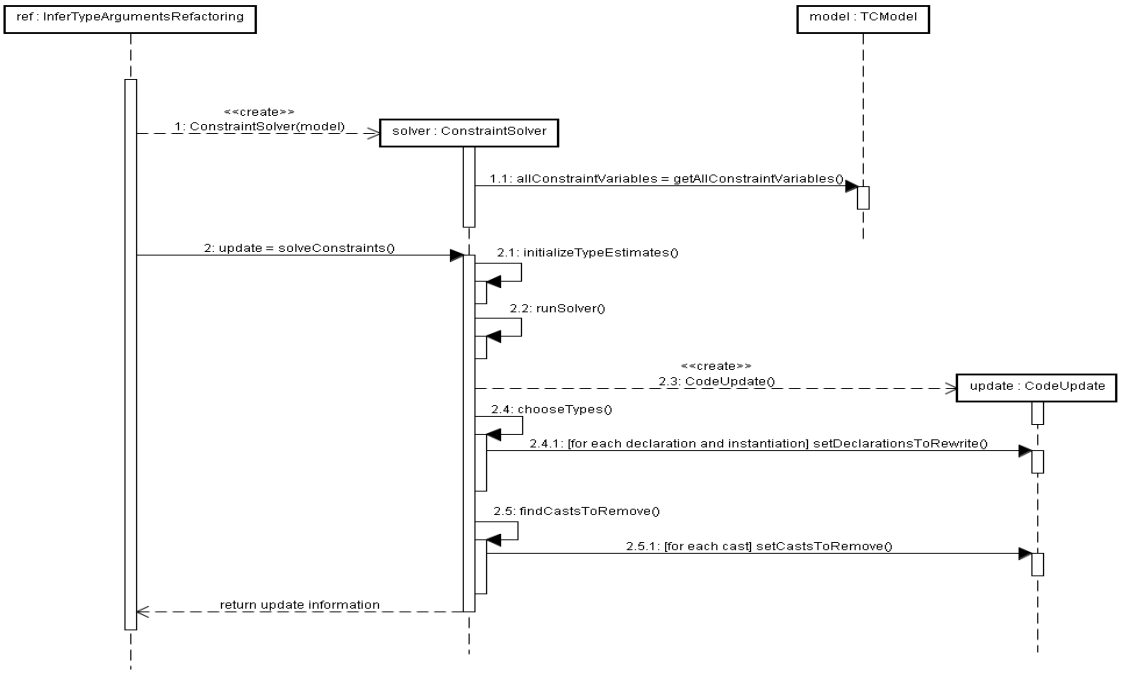

Figure 4.3: Algorithm overview - Phase 2

and constructors) or (ii) the type universe (in all other cases). Second, in step 2.2, the iterative phase is started with the execution of the method *runSolver*. In this step, a work-list is created with the constraint variables whose initial estimation has been recently associated. In each iteration, a constraint variable α_1 is removed from the work-list, and all type constraints that refer to α_1 are examined. For each type constraint $t \equiv \alpha_1 \leq \alpha_2$, the estimates associated with α_1 and α_2 are updated by removing any element that would violate t. If at least one element is removed (meaning that estimation has changed), all constraint variables related to α_1 and α_2 are reentered in the work-list in order to propagate the modified estimation.

Since the type constraint generation strategy is based on small steps (see Section 3.2) and estimations are finite sets per constraint variable, algebraic operations such as intersection can be performed directly during the inspection of a given type constraint. Take a look at Figure 4.4 as a simple example.

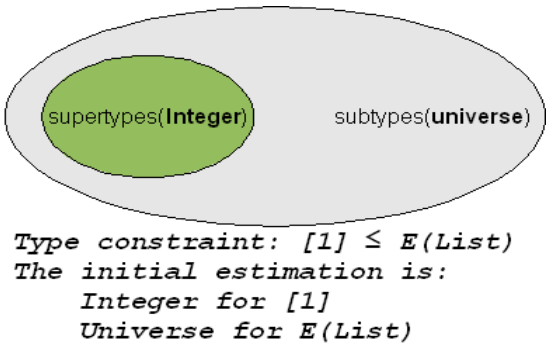

Figure 4.4: Intersection of constraint variable estimates

In this figure the type constraint $[1] \leq E(List)$ is being examined. Since the initial

estimation for [1] is *Integer* and for $E(List)$ is *universe*, the new estimation for $E(List)$ is: $supertypes(Integer) \cap subtypes(universe) \equiv supertypes(Integer)$. Note that, because of the subtype relationship ($\leq$) of the constraint, the intersection happens between supertypes of *Integer* and the subtypes of *universe*. If the relation were equals (=), the new estimation for $E(List)$ would be: $Integer \cap universe \equiv Integer$.

According to Fuhrer *et al.* (2005), estimates monotonically decrease in size as constraint solution progresses, therefore termination is guaranteed – since Chapter 3 only extends the original proposal to consider the use of aspects constructs, termination is also guaranteed for aspect-aware migration. The result of this process is a set of legal types for each constraint variable. Since there is a need to be a singleton type, one must be chosen. It is accomplished by the method *chooseTypes* in step 2.4 of Figure 4.3. Note that this method sets into the *CodeUpdate* object the type chosen for each constraint variable related to a generic class (step 2.4.1).

The optimization of this step is to select a type that maximizes the number of casts removed. As a simple approximation to this criterion, the algorithm selects an arbitrary most specific type from the current estimation. Even though this strategy is overly restrictive (a less specific type may suffice to remove the maximum number of casts), and potentially sub-optimal, the approach appears to be quite effective in practice (DONOVAN et al., 2004; FUHRER et al., 2005; DINCKLAGE; DIWAN, 2004).

Since this dissertation generates constraints for wildcards and the use of wildcards usually allows the removal of more insecure downcasts (when compared with a specific type), the *chooseTypes* method must be modified to give preference to wildcards over classes and interfaces. For example: ? *extends Number* precedes *Number*, which precedes ? *extends Object*, which precedes *Object*.

Similarly to step 2.4, the method *findCastsToRemove* (step 2.5) is the responsible to check – in each constraint variable that represents a cast – if the chosen type for the expression being casted can be assigned directly to the target type. If so, the cast is redundant, and then, it is marked to be removed (step 2.5.1).

4.2.3 Code Rewrite

Constraint solving phase yields a unique type for each constraint variable. Then, allocation and declaration sites that refer to generic classes must be rewritten if at least one of their inferred actual type parameters is more specific than existing declarations. Figure 4.5 overviews the main steps of this third and last phase of the algorithm.

For each CU, the code marked to be refactored in the *CodeUpdate* object is rewritten to add actual type parameters (step 2.1) and to remove insecure downcasts (step 2.2). Note that these casts can be safely removed because they become redundant with the new type parameters recently added. To support this refactoring in AO programs it is necessary to improve existing code rewriting framework to also work with aspect modules.

4.3 Discussion

This section aims to discuss some subtleties and advantages related to the generification proposed in this chapter. It includes a discussion about the main design decisions, the migration of the motivational examples and a parallel between the

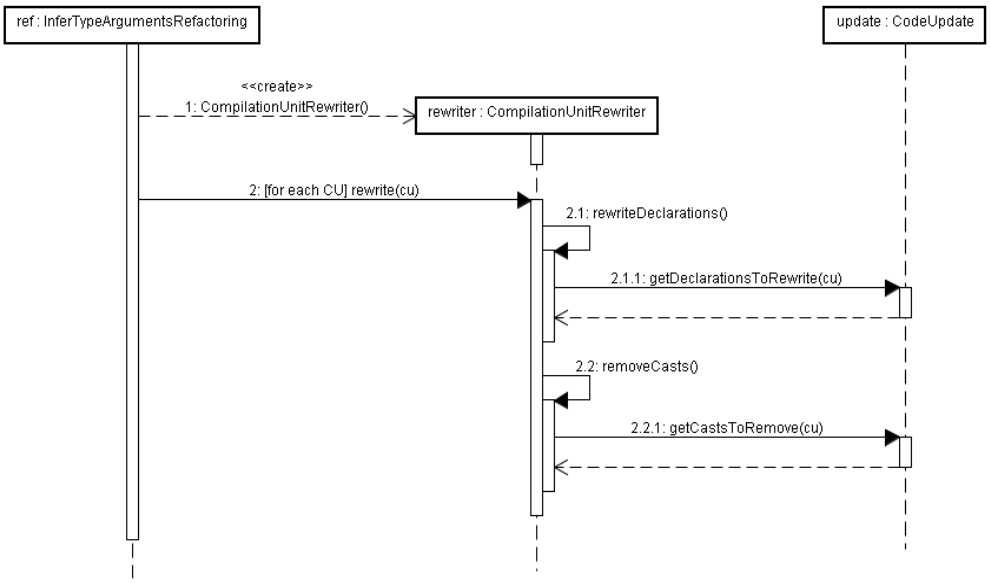

Figure 4.5: Algorithm overview - Phase 3

current state-of-the-art.

4.3.1 Design Decisions

There are some important design decisions that deserve a little of attention and therefore are described below.

4.3.1.1 Inference of Wildcards

First of all, it is worth to say that up to the time this dissertation was written, only two works (KIEŻUN et al., 2007; CRACIUN et al., 2009) claim to be able to infer wildcards.

Although Kieżun *et al.* (2007) allege being able to infer these types, they do not show the type constraint rules for such inference. Moreover, they do not make clear what kind of wildcard they are able to infer and if the inference of such types occurs only in the new parameterization process or if they have improved the existing instantiation solution they have built their work on.

In contrast, Craciun *et al.*'s (2009) study is focused on the variance of GJ type system. In other words, their proposal aims to provide a more flexible solution than existing works by inferring mainly wildcards for generic types. The solution uses an interval-based inference approach to infer wildcards for method's parameters based on how they are used into the method's body.

The generic migration presented in this dissertation is built on the same foundation of Kieżun *et al.*'s proposal. However, unlike their work, which does not show details about wildcard inference, the constraint generation rules presented in Chapter 3 explicitly show, in each rule, if the program construct being analyzed can declare or not a wildcard. This strategy makes clear where the type system allows the use of wildcards, what kind of wildcards are inferred and how these types are built.

4.3.1.2 Inference in Subclasses of Generic Classes

The Generic Java type system allows library designers to freely generify methods and classes independently of clients that define subclasses or subinterfaces of the library (GOSLING et al., 2005). Moreover, subclasses of generic classes that specify actual type parameters in overriding methods must instantiate the superclass accordingly (passing the appropriate type parameters) to become well-typed.

Considering these details, all type constraint generated by rules R15-R20 (see Section 3.2.3) must be discarded if no constraint that adds actual type parameters for the superclass is found.

Note that even though Fuhrer *et al.* (2005) and Kieżun *et al.* (2007) claim to be able to infer types in such constructs, they do not show the type constraint rules neither mention this important overriding restriction – which when not respected, causes the ill-typing of the hierarchy.

4.3.1.3 Signature Matching Pattern of Pointcuts

The proposed solution does not contemplate the refactoring of pointcut's signature matching pattern because it does not improve neither the type safety nor the source code readability (no casts can be removed). Moreover, since AspectJ also uses raw types to ensure backward compatibility, the option for not modifying the pointcut's signature also ensures that the advice matching remains unchanged after the generic migration – even when actual type parameters are added to joinpoints.

4.3.1.4 Behavior Preservation

The translation from non-generic to generic code must preserve the dynamic behavior of the program in all contexts. In particular, it must not throw different exceptions or differ in other observable respects. It must interoperate with existing modules in exactly the same way that the original code did. Therefore, the following three strategies are used in this work to ensure that the behavior will be preserved:

1. Similarly to other works (DONOVAN; ERNST, 2003; KIEŻUN et al., 2005), the strict notion of preserving behavior based on preserving program's erasure is applied. In other words, the compiled bytecode remains unchanged after the refactoring is finalized.

   Figure 4.6 shows, on the left hand side, both generic and non-generic versions of a given fragment of code and, on the right side, the bytecode[6] that represents the compiled version of these fragments. What is important to highlight in this figure is that both versions of the code when compiled generate the same bytecode. It happens because the actual type parameters (on the generic version) are the only differences between the fragments. Hence, since the type erasure process erase all actual type parameters, the resulting compiled version remains unchanged. Therefore, for pure Java code, preserving program's erasure in a generic migration is enough for preserving program's behavior;

2. When aspects are considered, the program's behavior can be affected if matched joinpoints receive new actual type parameters – the fragile pointcut problem. Since this work does not modify signature matching pattern

[6]Note that a more human readable syntax was used for the bytecode to simplify the reader comprehension.

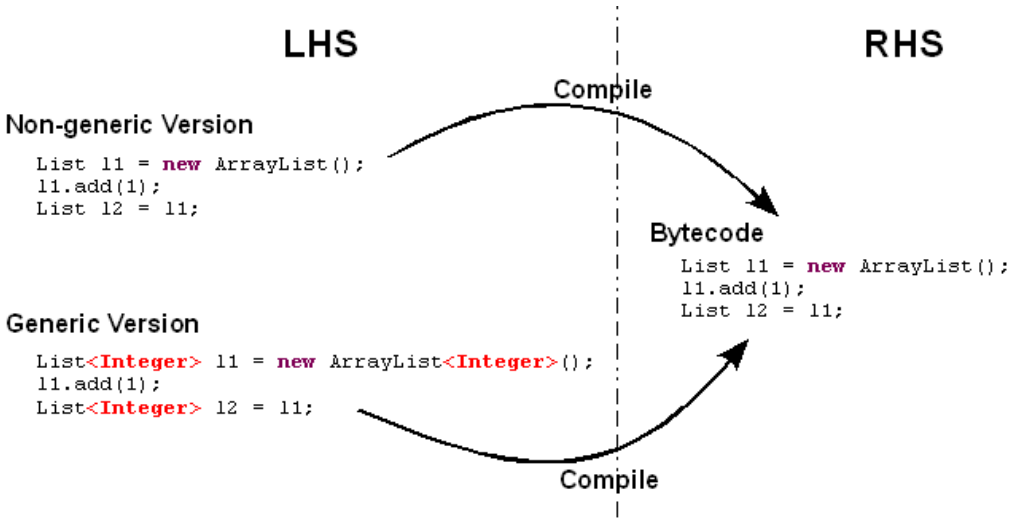

Figure 4.6: Bytecode remains unchanged.

(see Section 4.3.1.3), the backward compatibility is guaranteed by the raw types usage. This means that the behavior is preserved because the advice will be matching exactly the same joinpoints that it was matching before the refactoring started;

3. Note that *args*(..) pointcut's primitive and *after*() *returning*(..) advices can also change the behavior of the program if the types inferred for these constructs do not consider the type of the whole collection of joinpoints being matched. Since the constraint generation phase takes into account this situation, only valid types are inferred for these constructs. If it is not possible to find a type which is valid for all joinpoints being matched (i.e. the type of the whole collection of joinpoints is completely unconstrained), the algorithm leaves these generic declarations raw.

4.3.1.5 Generic Aspects

AspectJ allows the definition of type parameters in abstract aspects. On the other hand, it does not allow the weaving of such constructs; only concrete aspects with all of their type parameters fully defined are weaved. In other words, once an aspect is parameterized all instantiations must be provided. Since this is a parameterization problem and since parameterization is out of the scope of this work, this dissertation does not contemplate type parameters declared in abstract aspects.

4.3.2 Migrating Motivational Examples

Due to some AspectJ's compiler tooling limitations[7], the product of this dissertation does not include an implementation of the aspect-aware generic migration. Therefore, it was not possible to apply this approach to some real case study. To fill this gap, this section presents the migration of each motivational example discussed previously. Note that only a short discussion and the final result of the refactoring

[7]See the issue about AspectJ's AST support https://bugs.eclipse.org/bugs/show_bug.cgi?id=146528.

is presented. Then, the reader is encouraged to take a look at Appendix A for the detailed application of the algorithm steps to each example.

For the remainder of this section, the emphasized code represents actual type parameters added and casts (if it is the case) removed by the generic migration. Note that there is a number following each type and variable. This number was used to disambiguate each declaration, so that it is possible to differentiate each type constraint created during the constraint generation phase depicted in Appendix A.

Poor Inference

Since a poor inference is not desired in the generic migration of AspectJ programs, the framework of type constraint presented in Chapter 3 was designed in such a way that parent declarations and inter-type declarations are considered during the constraint generation phase. Therefore, applying the algorithm previously discussed in this chapter to example of Listing 4.1, the resulting code is the following:

```
1   class C{
2       void m1(List1<? extends Number>  l2){ .. }
3       public static void main(String[] args) {
4           List3<Integer> l4 = new ArrayList5<Integer>();
5           l4.add(1);
6           new C().m1(l4);
7       }
8   }
9   aspect A{
10      void C.m2(){
11          List6<Double> l7 = new ArrayList8<Double>();
12          l7.add(1.1);
13          m1(l7);
14      }
15  }
```

In this case, the algorithm has considered two calls to method $m1$: (i) the virtual call $new\ C().m1(l_4)$ at line 6 and; (ii) the local call $m1(l_7)$ at line 13. Since one call is passing a list of integers and another is passing a list of doubles, the algorithm rewrote the declaration of method $m1$ to accept a list of any type that extends *Number*.

Note that the problem of poor inference is resolved because it improves (with generic type annotations) the code inside of aspect A and accommodates the local method call from the inter-type declaration (line 13). For more details about the constraint generation and the application of the algorithm over this example, take a look on Appendix A.1.

Ill-typing

As it was discussed previously, all overriding relationships – including those defined by aspects – must be considered during a generic migration, otherwise the refactored version of the code will be ill-typed.

```
1   class C1 { .. }
2   class C2 extends C1 {
3       void m(List1<Integer> l2){..}
4       public static void main(String[] args) {
5           List3<Integer> l4 = new ArrayList5<Integer>();
6           l4.add(1);
7           new C2().m1(l4);
8       }
```

```
9   }
10  aspect A{
11      void C1.m(List6<Integer> l7){..}
12  }
```

The generic migration proposed in this dissertation is capable to identify overriding relationships no matter if it is based on an aspect construction or not. That is the reason why the above code (i.e. the refactored version of Listing 4.2) contains *List⟨Integer⟩* as the parameter declaration of the implicitly overridden method *m* (lines 3 and 11).

Note that the hierarchy where *C2 extends C1* is only well-typed when the method declared at line 3 and the method declared at line 11 have exactly the same formal parameter. Refer to Appendix A.2 for more details about the transformation from Listing 4.2 to the above code.

Pointcut's Fragility and Inference of Wildcards

An advice may execute before, after or around many different joinpoints. In these cases, the type of the advice is not directly connected with the type of a single joinpoint, but with the whole collection of joinpoints. To generate type constraints in such situations, the type for the entire collection must be defined, and then, linked to the type of the advice. Normally, the type of the collection (the pointcut) will be highly polymorphic, and the type of each element (joinpoint) will be less polymorphic.

Note that, during the code transformation, if the type of the entire collection of joinpoints is not linked to the type of the advice, the matching may change without warnings. Therefore, the task of adding actual type parameters to aspect-oriented programs must check all advices to verify if they are matching the piece of code being refactored.

```
1   class C{
2       void m1(List1<Integer> l2){..}
3       void m2(List3<Double> l4){..}
4
5       public static void main(String args[]){
6           List5<Integer> l6 = new ArrayList7<Integer>();
7           l6.add(1);
8           new C().m1(l6);
9
10          List8<Double> l29 = new ArrayList10<Double>();
11          l29.add(1.1);
12          new C().m2(l29);
13      }
14  }
15  aspect A{
16      before(List11<? extends Number> l12) : execution(void  C.m*(..))
17          && args(l12){..}
18  }
```

Knowing that GA subtyping is invariant and that the variance of type parameters is provided only by wildcards, the best choice for the *args*(..) construct (line 16) in the above refactored version of Listing 4.1.4 is *List11<? extends Number>* . Note that, in this example, the wildcard construction represents the type of the entire collection of joinpoints. In other words, this type keeps the advice matching both methods *m1* and *m2* even after class *C* be generificated. For more details about such inference, take a look on Appendix A.4.

Unsafe Type System

Unfortunately the AspectJ language does not have a safe type system (see Section 3.2.4). This means that current type rules are not able to ensure the type-correctness of a program when it uses around advices and/or proceed calls.

The solution proposed in this dissertation takes into account those cases, and therefore, provides type parameter annotations according to AspectJ Developers Notebook (2005). Take the example of the generic version of Listing 4.5 presented below.

```
1   class C<T extends Number> {
2       List1<T> m(List2<T> l3) { return l3; }
3   }
4   aspect A{
5       List4<Number> around(): execution(List m(..)) {
6           List5<Number> l6 = proceed();
7           l6.add(1);
8           return l6;
9       }
10  }
11  class Main {
12      public static void main(String args[]){
13          List7<Double> l8 = new ArrayList9<Double>();
14          l8.add(1.1);
15          List10<Double> l211  = new C<Double>().m(l8);
16          for (Iterator<Double> iter=l211.iterator(); iter.hasNext();){
17              Double d = (Double)iter.next();
18              ...
19          }
20      }
21  }
```

Even though AspectJ considers this code type-correct, the return of the advice declared at line 5 (i.e. $List_4\langle Number\rangle$) is not assignable to the return of method m (i.e. $List_1\langle T\rangle$) because T may be *Integer*, *Double* or any other subtype of *Number* at runtime. It is important to say that, in the current version of AspectJ (version 1.6.3), there is no way to make the above piece of code type-safe because the matching of advices always occurs against type variables erasure. Therefore, it is up to the programmer to avoid this kind of construction.

Take a look on Appendix A.5 for more details about how constraints are generated for unsafe constructions.

Others

Besides wildcards, there are other OO constructs that were not resolved in the main foundation of this work (FUHRER et al., 2005). These cases are mainly related to: raw types used in signature of methods declared into generic classes; calls for methods defined in the super generic class; and subclasses of generic classes with the superclass being instantiated accordingly (GOSLING et al., 2005).

The piece of code presented below shows the generic class $C2$ extending the generic class $C1$. Note that, even though method $m2$ (line 5) is declared into a generic class, its signature defines a raw $List_3$. Moreover, $C2$ is not passing the actual type parameter to its super generic class $C1$.

```
1   class C1<T extends Number>{
2       void m1(List1<T> l2){..}
3   }
4   class C2<T> extends C1{
5       void m2(List3 l4){
6           m1(l4);
7       }
8       void m3(List5<T> l6) {..}
9   }
10  class Main {
11      public static void main(String args[]){
12          List7 l8 = new ArrayList9();
13          l8.add(1);
14          new C2().m2(l8);
15      }
16  }
```

Although these problems seem to be caused by an incomplete parameterization – which is out of the scope of this work –, it is quite common in practice. Usually, method declarations without actual type parameter into generic classes are helper constructions and, super types not being instantiated accordingly normally occurs because the superclass was generificated without considering existing subclasses.

Therefore, the generic migration of class $C2$ must take into account at least the following three details: (i) even though `m2` was declared into a generic class, actual type parameters must be inferred for this method; (ii) the super method call at line 6 must also be considered; and (iii) the superclass $C1$ must be instantiated passing the appropriate type parameter.

```
1   class C1<T extends Number>{
2       void m1(List1<T> l2){..}
3   }
4   class C2<T> extends C1<Integer>{
5       void m2(List3<Integer> l4){
6           m1(l4);
7       }
8       void m3(List5<T> l6) {..}
9   }
10  class Main {
11      public static void main(String args[]){
12          List7<Integer> l8 = new ArrayList9<Integer>();
13          l8.add(1);
14          new C2().m2(l8);
15      }
16  }
```

The solution proposed in this dissertation understand these details and generate type constraints accordingly. That is why the above refactored version of the code contains actual type annotations in all declarations which refer to generic classes. Since a list of integers is being passed to $m2$ (line 14), the algorithm infers $List_3$<Integer> to $m2$'s parameter (line 5). Since $m2$ is calling $m1$ declared in $C1$, the algorithm infers $C1<Integer>$ to the superclass of $C2$ (line 4).

4.3.3 Parallel between Generic Migration Solutions

There exists in the literature of object-oriented refactoring many approaches to transform Java classes to its generic version (where the class definition specifies formal type parameters). This transformation is known as *parametrization problem.*

Also, some researches aim to solve the *instantiation problem*, which intent to rewrite all clients of generic classes to inform actual type parameters.

Duggan (1999) proposed a technique to translate monomorphic classes to parametric ones by inferring type arguments information (the parametrization problem). Although this is the first work in this area, Duggan's analysis leaves classes with excess of type parameters and its solution is not applicable to Java because the type system used differs from the current GJ (GOSLING et al., 2005) in several ways.

Donovan *et al.* (2004) proposed an algorithm based on type constraint rules to rewrite clients of generic classes to inform actual type parameters (the instantiation problem). They tried to ensure that the inferred types are as specific as possible, in order to minimize the number of insecure downcasts needed.

In parallel, Dincklage and Diwan (2004) proposed a new algorithm which included the parametrization and the instantiation process. Unfortunately, their solution was rejected by (FUHRER et al., 2005), since it does not preserve the program's behavior in some cases.

Looking at the problem from an efficient and accurate perspective, Tip *et al.* (2003) presented a study for migrating clients of the Java collection framework. Their solution mainly differs from (DONOVAN et al., 2004) in the sense that the algorithm is more scalable. As a natural evolution, Fuhrer *et al.* (2005) extended Tip *et al.*'s work to infer types for any generic construction and Kieżun *et al.* (2007) extended Fuhrer *et al.*'s work to include the parametrization of non-generic classes.

In the most recent study of this area, Craciun *et al.* (2009) use an interval-based inference approach focusing mainly on the inference of wildcards. According to the authors, the use of wildcards instead of specific types gives a more flexible solution than previous works which conservatively assume invariant subtyping.

Even though the idea of extending existing approaches to perform this kind of migration in the AO context has been previously suggested (HANNEMANN, 2006), it seems that a concrete proposal has not been done. Therefore, this dissertation have used the type constraint generation rules presented in Chapter 3 for extending Furher *et al.*'s (2005) proposal to consider the use of aspects to encapsulate crosscutting concerns and to infer wildcards when applicable.

Table 4.1: Parallel between Generic Migration Solutions

Work	**AspectJ**	**Parametrization problem**	**Instantiation problem**	**Infer wildcards?**	**Tool support**
(DUGGAN, 1999)		X			
(DONOVAN et al., 2004)			X		X
(DINCKLAGE; DIWAN, 2004)		X	X		X
(TIP et al., 2004)			X		X
(FUHRER et al., 2005)			X		X
(KIEŻUN et al., 2007)		X	X	X	X
(CRACIUN et al., 2009)			X	X	X
Proposed approach	X		X	X	

Finally, Table 4.1 presents a comparison of the current state-of-the-art related to generic code migration and the solution proposed in this chapter. Note that all these works aim to refactor the code to use generics in order to improve readability and the capability to check types at compile-time.

5 CONCLUSION

The parametric polymorphism in both Java and AspectJ improves the type safety and the expressiveness of the source code. However, to legacy non-generic code take advantage of this pervasive feature, it must be migrated to explicitly supply actual type parameters in both declarations and instantiations of generic classes. Such problem is known in the generic migration literature as the instantiation problem.

Since AOP uses separate modules to deal with crosscutting concerns and since the aspect weaving takes place just after the type erasure process, the task related to add actual type parameter to AspectJ programs is somewhat complex and error prone. Subtleties involving both the absence of type information, inter-type declarations, parent declarations, advices and pointcut expressions must be taken into account since they may affect the result of the refactoring. Therefore, tools to help this migration are essential to minimize the transformation effort and to prevent the introduction of new errors.

Even though the product of this dissertation does not include an implementation of an aspect-aware generic code migration, it provides the main steps to automatize this process. This study includes, in Chapter 3, a framework of type constraints for the polymorphic version of AspectJ and discusses, in Chapter 4, how this framework can be used with an existing algorithm to address the instantiation problem in both Java and AspectJ languages.

Section 4.1 presents several examples which claim for migration to take advantage of generic types. These examples, beyond illustrating the usefulness of the aspect-aware generic migration algorithm, also enabled us to verify that this proposal is viable and perfectly feasible in terms of implementation. In Section 4.3.2, these examples were refactored to include type parameter annotations and the details of this refactoring are shown in Appendix A. In order to ensure behavior preservation, the program's erasure was conserved. Moreover, the whole collection of matched join-points was faced as a unique type per advice and the pointcut's signature matching pattern was not modified.

This proposal distinguishes itself from its basis (FUHRER et al., 2005) in several important ways: it subsumes existing object-oriented support by providing actual type parameters for some constructs that were not considered in the original proposal; it infers wildcard type parameters when applicable; and considers the use of aspects to encapsulate crosscutting concerns.

Another relevant contribution is the aspect-aware type constraint framework itself. In an effort to show a practical example, this work presented the main steps to integrate this framework with an automated generic migration. However, other works may also get benefits from this proposal since type constraints are being

used to solve many different problems in the literature. Some examples are: type inference; type checking; refactorings that use pre- and post-conditions to guarantee program's behavior preservation; analysis of the Java bytecode; etc.

It is important to point out that during the conception of this dissertation, no other type constraint framework for the AspectJ language was found. Therefore, the presented solution seems to be the first to propose a set of inference rules for generating type constraints for AspectJ programs. It is also outstanding to note that, since the Generic AspectJ design is still open – there are some unresolved issues related to the matching against generic types –, this work contemplates only the current production ready implementation of AspectJ (version 1.6.3).

With aspect-oriented programming emerging as a powerful tool in the software development (JAGADEESAN et al., 2006), tools to help refactoring in this context are indispensable. Therefore, without an automated aspect-aware generic migration, most of the non-generic Java code which has at least one combined aspect is destined to "subsist" with unsafe types, unless, of course, a great effort is put into a manual migration.

5.1 Future Work

There are several opportunities for research in the areas covered by this dissertation. The following are possible continuations of this work:

Support for all AspectJ Constructs

The type constraint framework introduced in this dissertation handles the full Java language and the most important constructions of AspectJ. However, it would be interesting to complete this framework to fully support AspectJ by including designators not covered in this work (i.e. *this*(..), *target*(..), *cflow*(..) and etc).

Tool Support

Future work can focus on improving existing Eclipse's "Infer Generic Type Arguments" refactoring (KIEŻUN et al., 2005) to consider the use of aspect abstractions and wildcards. In that way, the aspect-aware generic migration can be applied in real world projects. This would help assess the advantages and disadvantages of the proposed approach and how it scales in large projects.

Improve other Refactorings to Support AspectJ Code

Future work may also focus on using the proposed aspect-aware generic type constraint framework to extend other refactorings to start considering aspects. Some options are: generalization (e.g., "Extract Interface" for re-routing the access to a class via a newly created interface and "Pull Up Members" for moving members into a superclass); parameterization of non-generic classes; and the customization of container classes.

Formal Proofs of Soundness and Completeness

It would be interesting if a future work provided proofs of soundness and completeness of the inference solution.

APPENDIX A

This appendix contains the details of the application of the aspect-aware generic code migration in the examples presented in Section 4.1. Each of the following sections includes: (i) the program to be refactored; (ii) the generation of all type constraints relative to the original program; (iii) the resolution of the most important constraints; and (iv) the resultant code – the generalized version.

For the remainder of this section, the emphasized code represents actual type parameters added and casts (if it is the case) removed by the migration. Note that there is a number following each type and variable. This number was used to differentiate each type constraint created during the constraint generation phase.

A.1 Poor Inference

Program to be Refactored

```
1       class C{
2               void m1(List1 l2){ .. }
3               public static void main(String[] args) {
4                       List3 l4 = new ArrayList5();
5                       l4.add(1);
6                       new C().m1(l4);
7               }
8       }
9       aspect A{
10              void C.m2(){
11                      List6 l7 = new ArrayList8();
12                      l7.add(1.1);
13                      m1(l7);
14              }
15      }
```

Phase 1 - Generation of Constraints

Line 2 - declaration: $List_1\ l_2$

$\xrightarrow{R1}$ RNonContextCgen([l_2], $List_1$, •)

$\xrightarrow{rn12}$ [l_2] = $List_1$ ∧ E([l_2]) = E($List_1$)

Line 4 - declaration: $List_3\ l_4$

$\xrightarrow{R1}$ RNonContextCgen([l_4], $List_3$, •)

$\xrightarrow{rn12}$ [l_4] = $List_3$ ∧ E([l_4]) = E($List_3$)

Line 4 - type of constructor call: new $ArrayList_5$()

$\xrightarrow{R12}$ RNonContextCgen([new $ArrayList_5$()], $ArrayList_5$, •)

$\xrightarrow{rn1}$ [new $ArrayList_5$()] = $ArrayList_5$

Line 4 - assignment: l_4 = new $ArrayList_5$()

$\xrightarrow{R4}$ NonContextCgen([new $ArrayList_5$()], ≤, [l_4], •)

$\longrightarrow$NonContextCgen([new $ArrayList_5$()], $\leq$, $List_3$, $\bullet$)
$\xrightarrow{n12}$ [new $ArrayList_5$()] $\leq$ $List_3$ $\wedge$
(E([new $ArrayList_5$()]) = E($List_3$) $\vee$ E($List_3$) $\in$ Wild(E([new $ArrayList_5$()]))

Line 5 - virtual method call: l_4.add(1)
$\xrightarrow{R7}$ [l_4] $\leq$ List $\vee$ [l_4] $\leq$ Collection
$\xrightarrow{R8}$ RContextCgen([l_4.add(1)], boolean, l_4, $\circ$)
$\xrightarrow{rc1}$ [l_4.add(1)] = boolean
$\xrightarrow{R9}$ ContextCgen([1], $\leq$, E, l_4, $\bullet$)
$\xrightarrow{c4}$ [1] $\leq$ E([l_4])

Line 6 - type of constructor call: new C()
$\xrightarrow{R12}$ RNonContextCgen([new C()], C, $\bullet$)
$\xrightarrow{rn1}$ [new C()] = C

Line 6 - virtual method call: new C().m1(l_4)
$\xrightarrow{R7}$ [new C()] $\leq$ C
$\xrightarrow{R9}$ ContextCgen([l_4], $\leq$, $List_1$, new C(), $\bullet$)
$\xrightarrow{c12}$ [l_4] $\leq$ $List_1$ $\wedge$ (E([l_4]) = E($List_1$) $\vee$ E($List_1$) $\in$ Wild(E([l_4])))

Line 11 - declaration: $List_6$ l_7
$\xrightarrow{R1}$ RNonContextCgen([l_7], $List_6$, $\bullet$)
$\xrightarrow{rn12}$ [l_7] = $List_6$ $\wedge$ E([l_7]) = E($List_6$)

Line 11 - type of constructor call: new $ArrayList_8$()
$\xrightarrow{R12}$ RNonContextCgen([new $ArrayList_8$()], $ArrayList_8$, $\bullet$)
$\xrightarrow{rn1}$ [new $ArrayList_8$()] = $ArrayList_8$

Line 11 - assignment: l_7 = new $ArrayList_8$()
$\xrightarrow{R4}$ NonContextCgen([new $ArrayList_8$()], $\leq$, [l_7], $\bullet$)
$\longrightarrow$NonContextCgen([new $ArrayList_8$()], $\leq$, $List_6$, $\bullet$)
$\xrightarrow{n12}$ [new $ArrayList_8$()] $\leq$ $List_6$ $\wedge$
(E([new $ArrayList_8$()]) = E($List_6$) $\vee$ E($List_6$) $\in$ Wild(E([new $ArrayList_8$()])))

Line 12 - virtual method call: l_7.add(1.1)
$\xrightarrow{R7}$ [l_7] $\leq$ List $\vee$ [l_7] $\leq$ Collection
$\xrightarrow{R8}$ RContextCgen([l_7.add(1.1)], boolean, l_7, $\circ$)
$\xrightarrow{rc1}$ [l_7.add(1.1)] = boolean
$\xrightarrow{R9}$ ContextCgen([1.1], $\leq$, E, l_7, $\bullet$)
$\xrightarrow{c4}$ [1.1] $\leq$ E([l_7])

Line 13 - local method call: m1(l_7)
$\xrightarrow{R11}$ NonContextCgen([l_7], $\leq$, $List_1$, $\bullet$)
$\xrightarrow{n12}$ [l_7] $\leq$ $List_1$ $\wedge$ (E([l_7]) = E($List_1$) $\vee$ E($List_1$) $\in$ Wild(E([l_7])))

The summary of the constraints generated in this first phase of the algorithm is presented in Table A.1.

Phase 2 - Constraints Solving

Important constraints

$[1] \leq E([l_4])$
$[1.1] \leq E([l_7])$
$E([l_4]) = E(List_1) \quad \vee \quad E(List_1) \in Wild(E([l_4]))$
$E([l_7]) = E(List_1) \quad \vee \quad E(List_1) \in Wild(E([l_7]))$
...

2.1 - Initializing type estimates

$[1] \equiv Integer$
$[1.1] \equiv Double$

Table A.1: Poor Inference - Final result of Phase 1

Code	**Type constraint(s)**	**Rule(s)**
$List_1\ \ l_2$	$[l_2] = List_1 \quad \wedge \quad E([l_2]) = E(List_1)$	R1
$List_3\ \ l_4$	$[l_4] = List_3 \quad \wedge \quad E([l_4]) = E(List_3)$	R1
$new\ \ ArrayList_5()$	$[new\ \ ArrayList_5()] = ArrayList_5$	R12
$l_4 = new\ \ ArrayList_5()$	$[new\ \ ArrayList_5()] \leq List_3 \quad \wedge$	R4
	$(E([new\ \ ArrayList_5()]) = E(List_3) \quad \vee \quad E(List_3)\ \in\ Wild(E([new\ \ ArrayList_5()])))$	
$l_4.add(1)$	$[l_4] \leq List \quad \vee \quad [l_4] \leq Collection$	R7
	$[l_4.add(1)] = boolean$	R8
	$[1] \leq E([l_4])$	R9
$new\ \ C()$	$[new\ \ C()] = C$	R12
$new\ \ C().m1(l_4)$	$[new\ \ C()] \leq C$	R7
	$[l_4] \leq List_1 \quad \wedge \quad (E([l_4]) = E(List_1) \quad \vee \quad E(List_1)\ \in\ Wild(E([l_4])))$	R9
$List_6\ \ l_7$	$[l_7] = List_6 \quad \wedge \quad E([l_7]) = E(List_6)$	R1
$new\ \ ArrayList_8()$	$[new\ \ ArrayList_8()] = ArrayList_8$	R12
$l_7 = new\ \ ArrayList_8()$	$[new\ \ ArrayList_8()] \leq List_6 \quad \wedge$	R4
	$(E([new\ \ ArrayList_8()]) = E(List_6) \quad \vee \quad E(List_6)\ \in\ Wild(E([new\ \ ArrayList_8()])))$	
$l_7.add(1.1)$	$[l_7] \leq List \quad \vee \quad [l_7] \leq Collection$	R7
	$[l_7.add(1.1)] = boolean$	R8
	$[1.1] \leq E([l_7])$	R9
$m1(l_7)$	$[l_7] \leq List_1 \quad \wedge \quad (E([l_7]) = E(List_1) \quad \vee \quad E(List_1)\ \in\ Wild(E([l_7])))$	R11

$E([l_4]) \equiv < universe >$

$E([l_7]) \equiv < universe >$

$List_1 \equiv java.util.List$

$E(List_1) \equiv < universe >$

...

2.2 - Solving constraints

In the beginning of this phase all constraint variables are copied to a new work-list. Then, in each iteration, a constraint variable is removed from the work-list up until it be empty.

The below constraint resolution shows an example of one iteration where $E([l_4])$ was removed from the work-list.

```
Evaluating constraint variable: E([l4])
  Type constraint related to E([l4]): [1] ≤ E([l4])
  Current estimation: [1]      ≡ Integer
                      E([l4])  ≡ < universe >
  Intersection between:
      superTypes(Integer) AND subTypes(< universe >)
  results in the following new estimation for E([l4]):
      superTypes(Integer)
  Type equivalence related to E([l4]): {E(List3), E(List1), ...}
      work − list += {E(List3), E(List1), ...}
```

Since the estimation of $E([l_4])$ has changed, the whole set of type equivalence must reenter in the work-list to be re-evaluated, so that, the new estimation can be propagated.

The below constraint resolution represents another iteration and it is evaluating constraint variable $E([l_7])$.

Evaluating constraint variable: $E([l_7])$
  Type constraint related to $E([l_7])$: $[1.1] \leq E([l_7])$
  Current estimation: $[1.1] \equiv Double$
  $E([l_7]) \equiv < universe >$
  Intersection between:
    $superTypes(Double)$ AND $subTypes(< universe >)$
  results in the following new estimation for $E([l_7])$:
    $superTypes(Double)$
  Type Equivalence related to $E([l_7])$: $\{E(List_6),\ E(List_1),\ \ldots\}$
    $work-list\ +=\ \{E(List_6),\ E(List_1),\ \ldots\}$

Similarly to the first iteration, the estimation of $E([l_7])$ has changed. Therefore, the whole set of type equivalence must reenter in the work-list to be re-evaluated.

...

2.3 - Choosing types

Since the final estimation for $E([l_4])$ is $superTypes(Integer)$, the single type chosen is:

$$E([l_4]) = Integer$$

Since the final estimation for $E([l_7])$ is $superTypes(Double)$, the single type chosen is:

$$E([l_7]) = Double$$

Since there exists the following constraints:

$$E([l_4]) = E(List_1) \quad \vee \quad E(List_1) \in Wild(E([l_4])) \text{ and}$$
$$E([l_7]) = E(List_1) \quad \vee \quad E(List_1) \in Wild(E([l_7]))$$

the intersection between

$$Integer = E(List_1) \quad \vee \quad E(List_1) \in Wild(Integer) \text{ and}$$
$$Double = E(List_1) \quad \vee \quad E(List_1) \in Wild(Double)$$

results in

$$E(List_1) \in Wild(Number)$$

Therefore the single type chosen is:

$$E(List_1) = ?\ extends\ Number$$

...

2.4 - Find casts to remove

There is no cast to be removed in this example.

Phase 3 - Code Rewrite

```
1       class C{
2               void m1(List1<? extends Number>  l2){ .. }
3               public static void main(String[] args) {
4                       List3<Integer> l4 = new ArrayList5<Integer>();
5                       l4.add(1);
6                       new C().m1(l4);
7               }
8       }
```

```
9       aspect A{
10              void C.m2(){
11                      List6<Double> l7 = new ArrayList8<Double>();
12                      l7.add(1.1);
13                      m1(l7);
14              }
15      }
```

A.2 Ill-typing

Program to be Refactored

```
1       class C1 { .. }
2       class C2 extends C1 {
3               void m(List1 l2){..}
4               public static void main(String[] args) {
5                       List3 l4 = new ArrayList5();
6                       l4.add(1);
7                       new C2().m1(l4);
8               }
9       }
10      aspect A{
11              void C1.m(List6 l7){..}
12      }
```

Phase 1 - Generation of Constraints

Line 3 - declaration: $List_1\ l_2$

$\xrightarrow{R1}$ RNonContextCgen([l_2], $List_1$, •)

$\xrightarrow{rn12}$ [l_2] = $List_1$ ∧ E([l_2]) = E($List_1$)

Line 5 - declaration: $List_3\ l_4$

$\xrightarrow{R1}$ RNonContextCgen([l_4], $List_3$, •)

$\xrightarrow{rn12}$ [l_4] = $List_3$ ∧ E([l_4]) = E($List_3$)

Line 5 - type of constructor call: new $ArrayList_5$()

$\xrightarrow{R12}$ RNonContextCgen([new $ArrayList_5$()], $ArrayList_5$, •)

$\xrightarrow{rn1}$ [new $ArrayList_5$()] = $ArrayList_5$

Line 5 - assignment: l_4 = new $ArrayList_5$()

$\xrightarrow{R4}$ NonContextCgen([new $ArrayList_5$()], ≤, [l_4], •)

⟶NonContextCgen([new $ArrayList_5$()], ≤, $List_3$, •)

$\xrightarrow{n12}$ [new $ArrayList_5$()] ≤ $List_3$ ∧
(E([new $ArrayList_5$()]) = E($List_3$) ∨ E($List_3$) ∈ Wild(E([new $ArrayList_5$()]))

Line 6 - virtual method call: l_4.add(1)

$\xrightarrow{R7}$ [l_4] ≤ List ∨ [l_4] ≤ Collection

$\xrightarrow{R8}$ RContextCgen([l_4.add(1)], boolean, l_4, ○)

$\xrightarrow{rc1}$ [l_4.add(1)] = boolean

$\xrightarrow{R9}$ ContextCgen([1], ≤, E, l_4, •)

$\xrightarrow{c4}$ [1] ≤ E([l_4])

Line 7 - type of constructor call: new C2()

$\xrightarrow{R12}$ RNonContextCgen([new C2()], C2, •)

$\xrightarrow{rn1}$ [new C2()] = C2

Line 7 - virtual method call: new C2().m1(l_4)

$\xrightarrow{R7}$ [new C2()] ≤ C2

$\xrightarrow{R9}$ ContextCgen([l_4], ≤, $List_1$, new C2(), •)

$\xrightarrow{c12}$ [l_4] ≤ $List_1$ ∧ (E([l_4]) = E($List_1$) ∨ E($List_1$) ∈ Wild(E([l_4])))

Line 11 - declaration: $List_6\ \ l_7$
$\xrightarrow{R1}$ RNonContextCgen([l_7], $List_6$, •)
$\xrightarrow{rn12}$ [l_7] = $List_6$ ∧ E([l_7]) = E($List_6$)

Line 3 and 11 - method overriding: C2.m($List_1\ \ l_2$) overrides C1.m($List_6\ \ l_7$)
$\xrightarrow{R16}$ RContextCgen([$List_1$], $List_6$, C2, •)
$\xrightarrow{rc13}$ [$List_1$] = $List_6$ ∧ E($List_1$) = E($List_6$)

The summary of the constraint generated in this first phase of the algorithm is provided in Table A.2.

Table A.2: Ill-typing - Final result of Phase 1

Code	**Type constraint(s)**	**Rule(s)**
$List_1\ \ l_2$	$[l_2] = List_1 \quad \wedge \quad E([l_2]) = E(List_1)$	R1
$List_3\ \ l_4$	$[l_4] = List_3 \quad \wedge \quad E([l_4]) = E(List_3)$	R1
$new\ \ ArrayList_5()$	$[new\ \ ArrayList_5()] = ArrayList_5$	R12
$l_4 = new\ \ ArrayList_5()$	$[new\ \ ArrayList_5()] \leq List_3 \quad \wedge$	R4
	$(E([new\ \ ArrayList_5()]) = E(List_3) \quad \vee \quad E(List_3) \in Wild(E([new\ \ ArrayList_5()])))$	
$l_4.add(1)$	$[l_4] \leq List \quad \vee \quad [l_4] \leq Collection$	R7
	$[l_4.add(1)] = boolean$	R8
	$[1] \leq E([l_4])$	R9
$new\ \ C2()$	$[new\ \ C2()] = C2$	R12
$new\ \ C2().m1(l_4)$	$[new\ \ C2()] \leq C2$	R7
	$[l_4] \leq List_1 \quad \wedge \quad (E([l_4]) = E(List_1) \quad \vee \quad E(List_1) \in Wild(E([l_4])))$	R9
$List_6\ \ l_7$	$[l_7] = List_6 \quad \wedge \quad E([l_7]) = E(List_6)$	R1
$C2.m(List_1\ \ l_2)$	$[List_1] = List_6 \quad \wedge \quad E(List_1) = E(List_6)$	R16
$overrides$		
$C1.m(List_6\ \ l_7)$		

Phase 2 - Constraints Solving

Important constraints

$[1] \leq E([l_4])$
$E([l_4]) = E(List_1) \quad \vee \quad E(List_1) \in Wild(E([l_4]))$
$E(List_1) = E(List_6)$
...

2.1 - Initializing type estimates

$[1] \equiv Integer$
$E([l_4]) \equiv < universe >$
$List_1 \equiv java.util.List$
$E(List_1) \equiv < universe >$
$List_6 \equiv java.util.List$
$E(List_6) \equiv < universe >$
...

2.2 - Solving constraints

In the beginning of this phase all constraint variables are copied to a new work-list. Then, in each iteration, a constraint variable is removed from the work-list up until it be empty.

The below constraint resolution shows an example of one iteration where $E([l_4])$ was removed from the work-list.

```
Evaluating constraint variable: E([l4])
  Type constraint related to E([l4]): [1] ≤ E([l4])
  Current estimation: [1]      ≡ Integer
                      E([l4])  ≡ <universe>
  Intersection between:
      superTypes(Integer) AND subTypes(<universe>)
  results in the following new estimation for E([l4]):
      superTypes(Integer)
  Type equivalence related to E([l4]): {E(List1), E(List6), ...}
      work-list += {E(List1), E(List6), ...}
```

Since the estimation of $E([l_4])$ has changed, the whole set of type equivalence must reenter in the work-list to be re-evaluated, so that, the new estimation can be propagated.

...

2.3 - Choosing types

Since the final estimation for $E([l_4])$ is *superTypes*(*Integer*), the single type chosen is:

$$E([l_4]) = Integer$$

Since there exists the following constraints:

$$E([l_4]) = E(List_1) \quad \vee \quad E(List_1) \in Wild(E([l_4]))$$

then

$$Integer = E(List_1) \quad \vee \quad E(List_1) \in Wild(Integer)$$

therefore the single type chosen is:

$$E(List_1) = Integer$$

Since

$$E(List_1) = E(List_6)$$

the single type chosen is:

$$E(List_6) = Integer$$

...

2.4 - Find casts to remove

There is no cast to be removed in this example.

Phase 3 - Code Rewrite

```
1        class C1 { .. }
2        class C2 extends C1 {
3                void m(List1<Integer> l2){..}
4                public static void main(String[] args) {
5                        List3<Integer> l4 = new ArrayList5<Integer>();
6                        l4.add(1);
7                        new C2().m1(l4);
8                }
9        }
10       aspect A{
11               void C1.m(List6<Integer> l7){..}
12       }
```

A.3 Pointcut's Fragility

Program to be Refactored

```
1       class C{
2               void m1(List1 l2){..}
3               void m2(List3 l4){..}
4       }
5       aspect A{
6               before(List5 l6) : execution(void  C.*(..))
7                       && args(l6){ .. }
8       }
```

Phase 1 - Generation of Constraints

Line 2 - declaration: $List_1\ l_2$

$\xrightarrow{R1}$ RNonContextCgen([l_2], $List_1$, •)

$\xrightarrow{rn12}$ [l_2] = $List_1$ ∧ E([l_2]) = E($List_1$)

Line 3 - declaration: $List_3\ l_4$

$\xrightarrow{R1}$ RNonContextCgen([l_4], $List_3$, •)

$\xrightarrow{rn12}$ [l_4] = $List_3$ ∧ E([l_4]) = E($List_3$)

Line 6 - declaration: $List_5\ l_6$

$\xrightarrow{R1}$ RNonContextCgen([l_6], $List_5$, •)

$\xrightarrow{rn12}$ [l_6] = $List_5$ ∧ E([l_6]) = E($List_5$)

Line 7 - args: args(l_6) matches C.m1($List_1\ l_2$)

$\xrightarrow{R21}$ ArgsCgen([$List_1$], [l_6], ○)

$\xrightarrow{ar15}$ ([$List_1$] ≤ [l_6] ∨ [$List_1$] >? [l_6]) ∧
(E([l_6]) = E([$List_1$]) ∨ E([l_6]) ∈ Wild(E([$List_1$])))

Line 7 - args: args(l_6) matches C.m2($List_3\ l_4$)

$\xrightarrow{R21}$ ArgsCgen([$List_3$], [l_6], ○)

$\xrightarrow{ar15}$ ([$List_3$] ≤ [l_6] ∨ [$List_3$] >? [l_6]) ∧
(E([l_6]) = E([$List_3$]) ∨ E([l_6]) ∈ Wild(E([$List_3$])))

The summary of the constraint generated in this first phase of the algorithm is provided in Table A.3.

Phase 2 - Constraints Solving

Important constraints

$E([l_6]) = E([List_1]) \quad \vee \quad E([l_6]) \in Wild(E([List_1]))$

$E([l_6]) = E([List_3]) \quad \vee \quad E([l_6]) \in Wild(E([List_3]))$

...

Table A.3: Pointcut's Fragility - Final result of Phase 1

Code	**Type constraint(s)**	**Rule(s)**
$List_1\ \ l_2$	$[l_2] = List_1 \quad \wedge \quad E([l_2]) = E(List_1)$	R1
$List_3\ \ l_4$	$[l_4] = List_3 \quad \wedge \quad E([l_4]) = E(List_3)$	R1
$List_5\ \ l_6$	$[l_6] = List_5 \quad \wedge \quad E([l_6]) = E(List_5)$	R1
$args(l_6)\ \ matches$ $C.m1(List_1\ \ l_2)$	$([List_1] \leq [l_6] \quad \vee \quad [List_1] >_? [l_6]) \quad \wedge$ $(E([l_6]) = E([List_1]) \quad \vee \quad E([l_6]) \in Wild(E([List_1])))$	R21
$args(l_6)\ \ matches$ $C.m2(List_3\ \ l_4)$	$([List_3] \leq [l_6] \quad \vee \quad [List_3] >_? [l_6]) \quad \wedge$ $(E([l_6]) = E([List_3]) \quad \vee \quad E([l_6]) \in Wild(E([List_3])))$	R21

2.1 - Initializing type estimates

$E([l_6]) \equiv < universe >$

$List_1 \equiv java.util.List$

$E([List_1]) \equiv < universe >$

$List_3 \equiv java.util.List$

$E([List_3]) \equiv < universe >$

...

2.2 - Solving constraints

Nothing can be resolved because the example does not give enough information about the body of the methods $m1$ and $m2$, neither about the clients calling these methods.

2.3 - Choosing types

There is no type to be chosen because the estimation, at this point, for constraint variables that refer to type variable are all $< universe >$.

2.4 - Find casts to remove

There is no cast to be removed because no type information could be inferred.

Phase 3 - Code Rewrite

The code remains the same. In other words, no refactoring could be performed without knowing the the method's clients and method's body.

```
1       class C{
2               void m1(List1 l2){..}
3               void m2(List3 l4){..}
4       }
5       aspect A{
6               before(List5 l6) : execution(void  C.*(..))
7                       && args(l6){ .. }
8       }
```

A.4 Inference of Wildcards

Program to be Refactored

```
1       class C{
2             void m1(List_1 l_2){..}
3             void m2(List_3 l_4){..}
4
5              public static void main(String args[]){
6                     List_5 l_6 = new ArrayList_7();
7                     l_6.add(1);
8                     new C().m1(l_6);
9
10                    List_8 l2_9 = new ArrayList_10();
11                    l2_9.add(1.1);
12                    new C().m2(l2_9);
13            }
14      }
15      aspect A{
16             before(List_11 l_12) : execution(void  C.m*(..))
17                    && args(l_12){ .. }
18      }
```

Phase 1 - Generation of Constraints

Line 2 - declaration: $List_1\ l_2$

$\xrightarrow{R1}$ RNonContextCgen([l_2], $List_1$, •)

$\xrightarrow{rn12}$ [l_2] = $List_1$ ∧ E([l_2]) = E($List_1$)

Line 3 - declaration: $List_3\ l_4$

$\xrightarrow{R1}$ RNonContextCgen([l_4], $List_3$, •)

$\xrightarrow{rn12}$ [l_4] = $List_3$ ∧ E([l_4]) = E($List_3$)

Line 6 - declaration: $List_5\ l_6$

$\xrightarrow{R1}$ RNonContextCgen([l_6], $List_5$, •)

$\xrightarrow{rn12}$ [l_6] = $List_5$ ∧ E([l_6]) = E($List_5$)

Line 6 - type of constructor call: new $ArrayList_7$()

$\xrightarrow{R12}$ RNonContextCgen([new $ArrayList_7$()], $ArrayList_7$, •)

$\xrightarrow{rn1}$ [new $ArrayList_7$()] = $ArrayList_7$

Line 6 - assignment: l_6 = new $ArrayList_7$()

$\xrightarrow{R4}$ NonContextCgen([new $ArrayList_7$()], ≤, [l_6], •)

⟶NonContextCgen([new $ArrayList_7$()], ≤, $List_5$, •)

$\xrightarrow{n12}$ [new $ArrayList_7$()] ≤ $List_5$ ∧
(E([new $ArrayList_7$()]) = E($List_5$) ∨ E($List_5$) ∈ Wild(E([new $ArrayList_7$()]))

Line 7 - virtual method call: l_6.add(1)

$\xrightarrow{R7}$ [l_6] ≤ List ∨ [l_6] ≤ Collection

$\xrightarrow{R8}$ RContextCgen([l_6.add(1)], boolean, l_6, ○)

$\xrightarrow{rc1}$ [l_6.add(1)] = boolean

$\xrightarrow{R9}$ ContextCgen([1], ≤, E, l_6, •)

$\xrightarrow{c4}$ [1] ≤ E([l_6])

Line 8 - type of constructor call: new C()

$\xrightarrow{R12}$ RNonContextCgen([new C()], C, •)

$\xrightarrow{rn1}$ [new C()] = C

Line 8 - virtual method call: new C().m1(l_6)

$\xrightarrow{R7}$ [new C()] ≤ C

$\xrightarrow{R9}$ ContextCgen([l_6], ≤, $List_1$, new C(), •)

$\xrightarrow{c12}$ [l_6] ≤ $List_1$ ∧ (E([l_6]) = E($List_1$) ∨ E($List_1$) ∈ Wild(E([l_6])))

Line 10 - declaration: $List_8\ l2_9$

$\xrightarrow{R1}$ RNonContextCgen([$l2_9$], $List_8$, •)

$\xrightarrow{rn12}$ [$l2_9$] = $List_8$ ∧ E([$l2_9$]) = E($List_8$)

Line 10 - type of constructor call: new $ArrayList_{10}$()

$\xrightarrow{R12}$ RNonContextCgen([new $ArrayList_{10}$()], $ArrayList_{10}$, •)

$\xrightarrow{rn1}$ [new $ArrayList_{10}$()] = $ArrayList_{10}$

Line 10 - assignment: $l2_9$ = new $ArrayList_{10}()$
$\xrightarrow{R4}$ NonContextCgen([new $ArrayList_{10}()$], $\leq$, [$l2_9$], $\bullet$)
$\longrightarrow$NonContextCgen([new $ArrayList_{10}()$], $\leq$, $List_8$, $\bullet$)
$\xrightarrow{n12}$ [new $ArrayList_{10}()$] $\leq$ $List_8$ $\wedge$
(E([new $ArrayList_{10}()$]) = E($List_8$) $\vee$ E($List_8$) $\in$ Wild(E([new $ArrayList_{10}()$]))

Line 11 - virtual method call: $l2_9$.add(1.1)
$\xrightarrow{R7}$ [$l2_9$] $\leq$ List $\vee$ [$l2_9$] $\leq$ Collection
$\xrightarrow{R8}$ RContextCgen([$l2_9$.add(1.1)], boolean, $l2_9$, $\circ$)
$\xrightarrow{rc1}$ [$l2_9$.add(1,1)] = boolean
$\xrightarrow{R9}$ ContextCgen([1.1], $\leq$, E, $l2_9$, $\bullet$)
$\xrightarrow{c4}$ [1] $\leq$ E([$l2_9$])

Line 12 - type of constructor call: new C()
$\xrightarrow{R12}$ RNonContextCgen([new C()], C, $\bullet$)
$\xrightarrow{rn1}$ [new C()] = C

Line 12 - virtual method call: new C().m1($l2_9$)
$\xrightarrow{R7}$ [new C()] $\leq$ C
$\xrightarrow{R9}$ ContextCgen([$l2_9$], $\leq$, $List_3$, new C(), $\bullet$)
$\xrightarrow{c12}$ [$l2_9$] $\leq$ $List_3$ $\wedge$ (E([$l2_9$]) = E($List_3$) $\vee$ E($List_3$) $\in$ Wild(E([$l2_9$]))

Line 16 - declaration: $List_{11}$ l_{12}
$\xrightarrow{R1}$ RNonContextCgen([l_{12}], $List_{11}$, $\bullet$)
$\xrightarrow{rn12}$ [l_{12}] = $List_{11}$ $\wedge$ E([l_{12}]) = E($List_{11}$)

Line 17 - args: args(l_{12}) matches C.m1($List_1$ l_2)
$\xrightarrow{R21}$ ArgsCgen([$List_1$], [l_{12}], $\circ$)
$\xrightarrow{ar15}$ ([$List_1$] $\leq$ [l_{12}] $\vee$ [$List_1$] $>_?$ [l_{12}]) $\wedge$
(E([l_{12}]) = E([$List_1$]) $\vee$ E([l_{12}]) $\in$ Wild(E([$List_1$])))

Line 17 - args: args(l_{12}) matches C.m2($List_3$ l_4)
$\xrightarrow{R21}$ ArgsCgen([$List_3$], [l_{12}], $\circ$)
$\xrightarrow{ar15}$ ([$List_3$] $\leq$ [l_{12}] $\vee$ [$List_3$] $>_?$ [l_{12}]) $\wedge$
(E([l_{12}]) = E([$List_3$]) $\vee$ E([l_{12}]) $\in$ Wild(E([$List_3$])))

The summary of the constraint generated in this first phase of the algorithm is provided in Table A.4.

Phase 2 - Constraints Solving

Important constraints

$[1] \leq E([l_6])$
$[1.1] \leq E([l2_9])$
$E([l_6]) = E(List_1) \vee E(List_1) \in Wild(E([l_6]))$
$E([l2_9]) = E(List_3) \vee E(List_3) \in Wild(E([l2_9]))$
$E([l_{12}]) = E([List_1]) \vee E([l_{12}]) \in Wild(E([List_1]))$
$E([l_{12}]) = E([List_3]) \vee E([l_{12}]) \in Wild(E([List_3]))$
...

2.1 - Initializing type estimates

$[1] \equiv Integer$
$[1.1] \equiv Double$
$E([l_6]) \equiv < universe >$
$E([l2_9]) \equiv < universe >$
$E([l_{12}]) \equiv < universe >$

Table A.4: Inference of Wildcards - Final result of Phase 1

Code	**Type constraint(s)**	**Rule(s)**
$List_1\ \ l_2$	$[l_2] = List_1 \quad \wedge \quad E([l_2]) = E(List_1)$	R1
$List_3\ \ l_4$	$[l_4] = List_3 \quad \wedge \quad E([l_4]) = E(List_3)$	R1
$List_5\ \ l_6$	$[l_6] = List_5 \quad \wedge \quad E([l_6]) = E(List_5)$	R1
$new\ \ ArrayList_7()$	$[new\ \ ArrayList_7()] = ArrayList_7$	R12
$l_6 = new\ \ ArrayList_7()$	$[new\ \ ArrayList_7()] \leq List_5 \quad \wedge$	R4
	$(E([new\ \ ArrayList_7()]) = E(List_5) \quad \vee \quad E(List_5) \in Wild(E([new\ \ ArrayList_7()])))$	
$l_6.add(1)$	$[l_6] \leq List \quad \vee \quad [l_6] \leq Collection$	R7
	$[l_6.add(1)] = boolean$	R8
	$[1] \leq E([l_6])$	R9
$new\ \ C()$	$[new\ \ C()] = C$	R12
$new\ \ C().m1(l_6)$	$[new\ \ C()] \leq C$	R7
	$[l_6] \leq List_1 \quad \wedge \quad (E([l_6]) = E(List_1) \quad \vee \quad E(List_1) \ \in \ Wild(E([l_6])))$	R9
$List_8\ \ l2_9$	$[l2_9] = List_8 \quad \wedge \quad E([l2_9]) = E(List_8)$	R1
$new\ \ ArrayList_{10}()$	$[new\ \ ArrayList_{10}()] = ArrayList_{10}$	R12
$l2_9 = new\ \ ArrayList_{10}()$	$[new\ \ ArrayList_{10}()] \leq List_8 \quad \wedge$	R4
	$(E([new\ \ ArrayList_{10}()]) = E(List_8) \quad \vee \quad E(List_8) \in Wild(E([new\ \ ArrayList_{10}()])))$	
$l2_9.add(1.1)$	$[l2_9] \leq List \quad \vee \quad [l2_9] \leq Collection$	R7
	$[l2_9.add(1.1)] = boolean$	R8
	$[1.1] \leq E([l2_9])$	R9
$new\ \ C()$	$[new\ \ C()] = C$	R12
$new\ \ C().m1(l2_9)$	$[new\ \ C()] \leq C$	R7
	$[l2_9] \leq List_3 \quad \wedge \quad (E([l2_9]) = E(List_3) \quad \vee \quad E(List_3) \ \in \ Wild(E([l2_9])))$	R9
$List_{11}\ \ l_{12}$	$[l_{12}] = List_{11} \quad \wedge \quad E([l_{12}]) = E(List_{11})$	R1
$args(l_{12})\ \ matches$	$([List_1] \leq [l_{12}] \quad \vee \quad [List_1] >_? [l_{12}]) \quad \wedge$	R21
$C.m1(List_1\ \ l_2)$	$(E([l_{12}]) = E([List_1]) \quad \vee \quad E([l_{12}]) \in Wild(E([List_1])))$	
$args(l_{12})\ \ matches$	$([List_3] \leq [l_{12}] \quad \vee \quad [List_3] >_? [l_{12}]) \quad \wedge$	R21
$C.m2(List_3\ \ l_4)$	$(E([l_{12}]) = E([List_3]) \quad \vee \quad E([l_{12}]) \in Wild(E([List_3])))$	

$List_1 \equiv java.util.List$

$E([List_1]) \equiv < universe >$

$List_3 \equiv java.util.List$

$E([List_3]) \equiv < universe >$

...

2.2 - Solving constraints

In the beginning of this phase all constraint variables are copied to a new work-list. Then, in each iteration, a constraint variable is removed from the work-list up until it be empty.

The below constraint resolution shows an example of one iteration where $E([l_6])$ was removed from the work-list.

```
Evaluating constraint variable: E([l6])
  Type constraint related to E([l6]): [1] ≤ E([l6])
  Current estimation: [1]     ≡ Integer
                      E([l6]) ≡ < universe >
  Intersection between:
      superTypes(Integer) AND subTypes(< universe >)
```

results in the following new estimation for $E([l_6])$:
$superTypes(Integer)$
Type equivalence related to $E([l_6])$: $\{E(List_1),\ E(List_5),\ E([l_{12}]),\ \ldots\}$
$work-list\ +=\ \{E(List_1),\ E(List_5),\ E([l_{12}]),\ \ldots\}$

Since the estimation of $E([l_6])$ has changed, the whole set of type equivalence must reenter in the work-list to be re-evaluated, so that, the new estimation can be propagated.

The below constraint resolution represents another iteration and it is evaluating constraint variable $E([l2_9])$.

Evaluating constraint variable: $E([l2_9])$
Type constraint related to $E([l2_9])$: $[1.1] \leq E([l2_9])$
Current estimation: $[1.1] \equiv Double$
$E([l2_9]) \equiv <universe>$
Intersection between:
$superTypes(Double)$ AND $subTypes(<universe>)$
results in the following new estimation for $E([l2_9])$:
$superTypes(Double)$
Type equivalence related to $E([l2_9])$: $\{E(List_3),\ E(List_8),\ E([l_{12}]),\ \ldots\}$
$work-list\ +=\ \{E(List_3),\ E(List_8),\ E([l_{12}]),\ \ldots\}$

Similarly to the first iteration, the estimation of $E([l2_9])$ has changed. Therefore, the whole set of type equivalence must reenter in the work-list to be re-evaluated.

...

2.3 - Choosing types

Since the final estimation for $E([l_6])$ is $superTypes(Integer)$ and $E([l_6]) = E(List_1)$, the types chosen are:

$E([l_6]) = Integer$
$E(List_1) = Integer$

Since the final estimation for $E([l2_9])$ is $superTypes(Double)$ and $E([l2_9]) = E(List_3)$, the types chosen are:

$E([l2_9]) = Double$
$E(List_3) = Double$

Since there exists the following constraints:

$E([l_{12}]) = E([List_1]) \vee E([l_{12}]) \in Wild(E([List_1]))$ and
$E([l_{12}]) = E([List_3]) \vee E([l_{12}]) \in Wild(E([List_3]))$

the intersection between

$E([l_{12}]) = Integer \vee E([l_{12}]) \in Wild(Integer)$ and
$E([l_{12}]) = Double \vee E([l_{12}]) \in Wild(Double)$

results in

$E([l_{12}]) \in Wild(Number)$

Therefore the single type chosen is:

$E([l_{12}]) = ?\ extends\ Number$

2.4 - Find casts to remove

There is no cast to be removed in this example.

Phase 3 - Code Rewrite

```
1       class C{
2              void m1(List1<Integer> l2){..}
3              void m2(List3<Double> l4){..}
4
5               public static void main(String args[]){
6                      List5<Integer> l6 = new ArrayList7<Integer>();
7                      l6.add(1);
8                      new C().m1(l6);
9
10                     List8<Double> l29 = new ArrayList10<Double>();
11                     l29.add(1.1);
12                     new C().m2(l29);
13             }
14      }
15      aspect A{
16             before(List11<? extends Number> l12) : execution(void  C.m*(..))
17                     && args(l12){ .. }
18      }
```

A.5 Unsafe Type System

Program to be Refactored

```
1       class C<T extends Number> {
2              List1<T> m(List2<T> l3) { return l3; }
3       }
4       aspect A{
5              List4 around(): execution(List m(..)) {
6                      List5 l6 = proceed();
7                      l6.add(1);
8                      return l6;
9              }
10      }
11      class Main {
12             public static void main(String args[]){
13                     List7 l8 = new ArrayList9();
14                     l8.add(1.1);
15                     List10 l211  = new C().m(l8);
16                     for (Iterator iter=l211.iterator(); iter.hasNext();) {
17                             Double d = (Double)iter.next();
18                             ...
19                     }
20             }
21      }
```

Phase 1 - Generation of Constraints

Line 2 - declaration: $List_2$<T> l_3

$\xrightarrow{R1}$ RNonContextCgen([l_3], $List_2$<T>, •)

$\xrightarrow{rn9}$ [l_3] = $List_2$ ∧ RNonContextCgen(E([l_3]), T, •)

$\xrightarrow{rn4}$ [l_3] = $List_2$ ∧ E([l_3]) = T

Line 2 - return: return l_3

$\xrightarrow{R6}$ NonContextCgen([l_3], ≤, $List_1$<T>, •)

$\xrightarrow{n9}$ [l_3] ≤ $List_1$ ∧ NonContextCgen(E([l_3]), =, T, •)

$\xrightarrow{n4}$ [l_3] ≤ $List_1$ ∧ E([l_3]) = T

Line 5 - around advice: $List_4$ around()

$\xrightarrow{R23}$ [$List_4$] = Object ∨ AroundCgen([$List_4$], ≤, $List_1$<T>, •)

$\xrightarrow{ad9}$ [$List_4$] = Object $\vee$ ([$List_4$] $\leq$ $List_1$ $\wedge$ AroundCgen(E([$List_4$]), =, T, $\bullet$))
$\xrightarrow{ad4}$ [$List_4$] = Object $\vee$ ([$List_4$] $\leq$ $List_1$ $\wedge$ E([$List_4$]) = |T|)
$\longrightarrow$[$List_4$] = Object $\vee$ ([$List_4$] $\leq$ $List_1$ $\wedge$ E([$List_4$]) = Number)

Line 6 - declaration: $List_5$ l_6
$\xrightarrow{R1}$ RNonContextCgen([l_6], $List_5$, $\bullet$)
$\xrightarrow{rn12}$ [l_6] = $List_5$ $\wedge$ E([l_6]) = E($List_5$)

Line 6 - proceed call: proceed()
$\xrightarrow{R24}$ RNonContextCgen([proceed()], $List_4$, $\bullet$)
$\xrightarrow{rn12}$ [proceed()] = $List_4$ $\wedge$ E([proceed()]) = E($List_4$)

Line 6 - assignment: l_6 = proceed()
$\xrightarrow{R4}$ NonContextCgen([proceed()], $\leq$, [l_6], $\bullet$)
$\longrightarrow$NonContextCgen([proceed()], $\leq$, $List_5$, $\bullet$)
$\xrightarrow{n12}$ [proceed()] $\leq$ $List_5$ $\wedge$
(E([proceed()]) = E($List_5$) $\vee$ E($List_5$) $\in$ Wild(E([proceed()]))

Line 7 - virtual method call: l_6.add(1)
$\xrightarrow{R7}$ [l_6] $\leq$ List $\vee$ [l_6] $\leq$ Collection
$\xrightarrow{R8}$ RContextCgen([l_6.add(1)], boolean, l_6, $\circ$)
$\xrightarrow{rc1}$ [l_6.add(1)] = boolean
$\xrightarrow{R9}$ ContextCgen([1], $\leq$, E, l_6, $\bullet$)
$\xrightarrow{c4}$ [1] $\leq$ E([l_6])

Line 8 - return: return l_6
$\xrightarrow{R6}$ NonContextCgen([l_6], $\leq$, $List_4$, $\bullet$)
$\xrightarrow{n12}$ [l_6] $\leq$ $List_4$ $\wedge$ (E([l_6]) = E($List_4$) $\vee$ E($List_4$) $\in$ Wild(E([l_6])))

Line 13 - declaration: $List_7$ l_8
$\xrightarrow{R1}$ RNonContextCgen([l_8], $List_7$, $\bullet$)
$\xrightarrow{rn12}$ [l_8] = $List_7$ $\wedge$ E([l_8]) = E($List_7$)

Line 13 - type of constructor call: new $ArrayList_9$()
$\xrightarrow{R12}$ RNonContextCgen([new $ArrayList_9$()], $ArrayList_9$, $\bullet$)
$\xrightarrow{rn1}$ [new $ArrayList_9$()] = $ArrayList_9$

Line 13 - assignment: l_8 = new $ArrayList_9$()
$\xrightarrow{R4}$ NonContextCgen([new $ArrayList_9$()], $\leq$, [l_8], $\bullet$)
$\longrightarrow$NonContextCgen([new $ArrayList_9$()], $\leq$, $List_7$, $\bullet$)
$\xrightarrow{n12}$ [new $ArrayList_9$()] $\leq$ $List_7$ $\wedge$
(E([new $ArrayList_9$()]) = E($List_7$) $\vee$ E($List_7$) $\in$ Wild(E([new $ArrayList_9$()])))

Line 14 - virtual method call: l_8.add(1.1)
$\xrightarrow{R7}$ [l_8] $\leq$ List $\vee$ [l_8] $\leq$ Collection
$\xrightarrow{R8}$ RContextCgen([l_8.add(1.1)], boolean, l_8, $\circ$)
$\xrightarrow{rc1}$ [l_8.add(1,1)] = boolean
$\xrightarrow{R9}$ ContextCgen([1.1], $\leq$, E, l_8, $\bullet$)
$\xrightarrow{c4}$ [1] $\leq$ E([l_8])

Line 15 - declaration: $List_{10}$ $l2_{11}$
$\xrightarrow{R1}$ RNonContextCgen([$l2_{11}$], $List_{10}$, $\bullet$)
$\xrightarrow{rn12}$ [$l2_{11}$] = $List_{10}$ $\wedge$ E([$l2_{11}$]) = E($List_{10}$)

Line 15 - type of constructor call: new C()
$\xrightarrow{R12}$ RNonContextCgen([new C()], C, $\bullet$)
$\xrightarrow{rn1}$ [new C()] = C

Line 15 - virtual method call: new C().m(l_8)
$\xrightarrow{R7}$ [new C()] $\leq$ C
$\xrightarrow{R8}$ RContextCgen([new C().m(l_8)], $List_1$<T>, new C(), $\circ$)
$\xrightarrow{rc10}$ [new C().m(l_8)] = $List_1$ $\wedge$ RContextCgen(E([new C().m(l_8)]), T, new C(), $\bullet$)
$\xrightarrow{rc4}$ [new C().m(l_8)] = $List_1$ $\wedge$ E([new C().m(l_8)]) = T(new C())
$\xrightarrow{R9}$ ContextCgen([l_8], $\leq$, $List_2$<T>, new C(), $\bullet$)

$\xrightarrow{c9}$ [l_8] $\leq$ $List_2$ $\wedge$ ContextCgen(E([l_8]), =, T, new C(), •)
$\xrightarrow{c4}$ [l_8] $\leq$ $List_2$ $\wedge$ E([l_8]) = T(new C())

Line 15 - assignment: $l2_{11}$ = new C().m(l_8)
$\xrightarrow{R4}$ NonContextCgen([new C().m(l_8)], $\leq$, [$l2_{11}$], •)
$\longrightarrow$NonContextCgen([new C().m(l_8)], $\leq$, $List_{10}$, •)
$\xrightarrow{n12}$ [new C().m(l_8)] $\leq$ $List_{10}$ $\wedge$
(E([new C().m(l_8)]) = E($List_{10}$) $\vee$ E($List_{10}$) $\in$ Wild(E([new C().m(l_8)])))

Line 16 - declaration Iterator iter
$\xrightarrow{R1}$ RNonContextCgen([iter], Iterator, •)
$\xrightarrow{rn12}$ [iter] = Iterator $\wedge$ E([iter]) = E(Iterator)

Line 16 - virtual method call $l2_{11}$.iterator()
$\xrightarrow{R7}$ [$l2_{11}$] $\leq$ List $\wedge$ [$l2_{11}$] $\leq$ Collection
$\xrightarrow{R8}$ RContextCgen([$l2_{11}$.iterator()], Iterator<E1>, $l2_{11}$, ○)
$\xrightarrow{rc10}$ [$l2_{11}$.iterator()] = Iterator $\wedge$ RContextCgen(E([$l2_{11}$.iterator()]), E1, $l2_{11}$, •)
$\xrightarrow{rc4}$ [$l2_{11}$.iterator()] = Iterator $\wedge$ E([$l2_{11}$.iterator()]) = E1($l2_{11}$)
Note: in this rule (R8), $E1$ was used in place of E to disambiguate

Line 16 - assignment iter = $l2_{11}$.iterator()
$\xrightarrow{R4}$ NonContextCgen([$l2_{11}$.iterator()], $\leq$, [iter], •)
$\longrightarrow$NonContextCgen([$l2_{11}$.iterator()], $\leq$, Iterator, •)
$\xrightarrow{n12}$ [$l2_{11}$.iterator()] $\leq$ Iterator $\wedge$
(E([$l2_{11}$.iterator()]) = E(Iterator) $\vee$ E(Iterator) $\in$ Wild(E([$l2_{11}$.iterator()])))

Line 16 - virtual method call iter.hasNext()
$\xrightarrow{R7}$ [iter] $\leq$ Iterator
$\xrightarrow{R8}$ RContextCgen([iter.hasNext()], boolean, iter, ○)
$\xrightarrow{rc1}$ [iter.hasNext()] = boolean

Line 17 - declaration Double d
$\xrightarrow{R1}$ RNonContextCgen([d], Double, •)
$\xrightarrow{rn12}$ [d] = Double

Line 17 - virtual method call iter.next()
$\xrightarrow{R7}$ [iter] $\leq$ Iterator
$\xrightarrow{R8}$ RContextCgen([iter.next()], E, iter, ○)
$\xrightarrow{rc4}$ [iter.hasNext()] = wildbound(E(iter))

Line 17 - casting (Double)iter.next()
$\xrightarrow{R2}$ RNonContextCgen([(Double)iter.next()], Double, •)
$\xrightarrow{rn1}$ [(Double)iter.next()] = Double
$\xrightarrow{R3}$ NonContextCgen([iter.next()], $\geq$, Double, •)
$\xrightarrow{n1}$ [iter.next()] $\geq$ Double
$\longrightarrow$Double $\leq$ [iter.next()]

Line 17 - assignment d = (Double)iter.next()
$\xrightarrow{R4}$ NonContextCgen([(Double)iter.next()], $\leq$, [d], •)
$\longrightarrow$NonContextCgen([(Double)iter.next()], $\leq$, Double, •)
$\xrightarrow{n1}$ [(Double)iter.next()] $\leq$ Double

The summary of the constraint generated in this first phase of the algorithm is provided in Table A.5.

Phase 2 - Constraints Solving

Important constraints

Around advice

$E([List_4]) = Number$

$E([l_6]) = E(List_5)$

Table A.5: Unsafe Type System - Final result of Phase 1

Code	**Type constraint(s)**	**Rule(s)**
$List_2 < T > \ l_3$	$[l_3] = List_2 \quad \wedge \quad E([l_3]) = T$	R1
$return \ l_3$	$[l_3] \leq List_1 \quad \wedge \quad E([l_3]) = T$	R6
$List_4 \ around()$	$[List_4] = Object \quad \vee \quad ([List_4] \leq List_1 \quad \wedge \quad E([List_4]) = Number)$	R23
$List_5 \ l_6$	$[l_6] = List_5 \quad \wedge \quad E([l_6]) = E(List_5)$	R1
$proceed()$	$[proceed()] = List_4 \quad \wedge \quad E([proceed()]) = E(List_4)$	R24
$l_6 = proceed()$	$[proceed()] \leq List_5 \quad \wedge$	R4
	$(E([proceed()]) = E(List_5) \quad \vee \quad E(List_5) \in Wild(E([proceed()])))$	
$l_6.add(1)$	$[l_6] \leq List \quad \vee \quad [l_6] \leq Collection$	R7
	$[l_6.add(1)] = boolean$	R8
	$[1] \leq E([l_6])$	R9
$return \ l_6$	$[l_6] \leq List_4 \quad \wedge \quad (E([l_6]) = E(List_4) \quad \vee \quad E(List_4) \in Wild(E([l_6])))$	R6
$List_7 \ l_8$	$[l_8] = List_7 \quad \wedge \quad E([l_8]) = E(List_7)$	R1
$new \ ArrayList_9()$	$[new \ ArrayList_9()] = ArrayList_9$	R12
$l_8 = new \ ArrayList_9()$	$[new \ ArrayList_9()] \leq List_7 \quad \wedge$	R4
	$(E([new \ ArrayList_9()]) = E(List_7) \quad \vee \quad E(List_7) \in Wild(E([new \ ArrayList_9()])))$	
$l_8.add(1.1)$	$[l_8] \leq List \quad \vee \quad [l_8] \leq Collection$	R7
	$[l_8.add(1.1)] = boolean$	R8
	$[1.1] \leq E([l_8])$	R9
$List_{10} \ l2_{11}$	$[l2_{11}] = List_{10} \quad \wedge \quad E([l2_{11}]) = E(List_{10})$	R1
$new \ C()$	$[new \ C()] = C$	R12
$new \ C().m(l_8)$	$[new \ C()] \leq C$	R7
	$[new \ C().m(l_8)] = List_1 \quad \wedge \quad E([new \ C().m(l_8)]) = T(new \ C())$	R8
	$[l_8] \leq List_2 \quad \wedge \quad E([l_8]) = T(new \ C())$	R9
$l2_{11} = new \ C().m(l_8)$	$[new \ C().m(l_8)] \leq List_{10} \quad \wedge$	R4
	$(E([new \ C().m(l_8)]) = E(List_{10}) \quad \vee \quad E(List_{10}) \in Wild(E([new \ C().m(l_8)])))$	
$Iterator \ iter$	$[iter] = Iterator \quad \wedge \quad E([iter]) = E(Iterator)$	R1
$l2_{11}.iterator()$	$[l2_{11}] \leq List \quad \wedge \quad [l2_{11}] \leq Collection$	R7
	$[l2_{11}.iterator()] = Iterator \quad \wedge \quad E([l2_{11}.iterator()]) = E1(l2_{11})$	R8
$iter = l2_{11}.iterator()$	$[l2_{11}.iterator()] \leq Iterator \quad \wedge$	R4
	$(E([l2_{11}.iterator()]) = E(Iterator) \quad \vee \quad E(Iterator) \in Wild(E([l2_{11}.iterator()])))$	
$iter.hasNext()$	$[iter] \leq Iterator$	R7
	$[iter.hasNext()] = boolean$	R8
$Double \ d$	$[d] = Double$	R1
$iter.next()$	$[iter] \leq Iterator$	R7
	$[iter.hasNext()] = wildbound(E(iter))$	R8
$(Double)iter.next()$	$[(Double)iter.next()] = Double$	R2
	$Double \leq [iter.next()]$	R3
$d = (Double)iter.next()$	$[(Double)iter.next()] \leq Double$	R4

$E([proceed()]) = E(List_4)$

$E([proceed()]) = E(List_5) \quad \vee \quad E(List_5) \in Wild(E([proceed()]))$

$[1] \leq E([l_6])$

$E([l_6]) = E(List_4) \quad \vee \quad E(List_4) \in Wild(E([l_6]))$

...

Main method

$E([l_8]) = E(List_7)$

$[1.1] \leq E([l_8])$

$E([l2_{11}]) = E(List_{10})$

$E([new\ C().m(l_8)]) = T(new\ C())$

$E([l_8]) = T(new\ C())$

$E([new\ C().m(l_8)]) = E(List_{10}) \quad \vee \quad E(List_{10}) \in Wild(E([new\ C().m(l_8)]))$

$E([iter]) = E(Iterator)$

$E([l2_{11}.iterator()]) = E1(l2_{11})$

$E([l2_{11}.iterator()]) = E(Iterator) \quad \vee \quad E(Iterator) \in Wild(E([l2_{11}.iterator()]))$

$[iter.hasNext()] = wildbound(E(iter))$

$[(Double)iter.next()] = Double$

$[(Double)iter.next()] \leq Double$

...

2.1 - Initializing type estimates

Around advice

$List_4 \equiv java.util.List$

$E([List_4]) \equiv < universe >$

$Number \equiv Number$

$E([l_6]) \equiv < universe >$

$[1] \equiv Integer$

...

Main method

$E([l_8]) \equiv < universe >$

$[1.1] \equiv Double$

$T(new\ C()) \equiv < universe >$

$E([l2_{11}]) \equiv < universe >$

$E([iter]) \equiv < universe >$

...

2.2 - Solving constraints

In the beginning of this phase all constraint variables are copied to a new work-list. Then, in each iteration, a constraint variable is removed from the work-list up until it be empty.

Around advice

The below constraint resolution shows an example of one iteration where $E([List_4])$ was removed from the work-list.

```
Evaluating constraint variable: E([List4])
  Type constraint related to E([List4]): E([List4]) = Number
  Current estimation: Number ≡ Number
```

```
                    E([List4]) ≡ < universe >
  Intersection between:
      Number AND < universe >
  results in the following new estimation for E([List4]):
      Number
  Type equivalence related to E([List4]): {E([proceed()]), E(l6), ...}
      work − list += {E([proceed()]), E(l6), ...}
```

Since the estimation of $E([List_4])$ has changed, the whole set of type equivalence must reenter in the work-list to be re-evaluated, so that, the new estimation can be propagated.

The below constraint resolution represents another iteration and it is evaluating constraint variable $E([l_6])$.

```
Evaluating constraint variable: E([l6])
  Type constraint related to E([l6]): [1] ≤ E([l6])
  Current estimation: [1] ≡ Integer
                      E([l6]) ≡ < universe >
  Intersection between:
      superTypes(Integer) AND subTypes(< universe >)
  results in the following new estimation for E([l6]):
      superTypes(Integer)
  Type equivalence related to E([l6]): {E(List5), E(List4), ...}
      work − list += {E(List5), E(List4), ...}


  Type constraint related to E([l6]): E([l6]) = E(List4) ∨ E(List4) ∈ Wild(E([l6]))
  Current estimation: E([l6]) ≡ superTypes(Integer)
                      E(List4) ≡ Number
  Intersection between:
      superTypes(Integer) AND Number
  results in the following new estimation for E([l6]):
      Number
  Type equivalence related to E([l6]): {E(List5), E(List4), ...}
      work − list += {E(List5), E(List4), ...}
```

The main difference from the other examples is that this constraint variable are being used in two type constraints. Therefore, for two constraint resolution were necessary.

...

Main method

The below constraint resolution represents another iteration and it is evaluating constraint variable $E([l_8])$.

```
Evaluating constraint variable: E([l8])
  Type constraint related to E([l8]): [1] ≤ E([l8])
  Current estimation: [1] ≡ Integer
                      E([l8]) ≡ < universe >
  Intersection between:
      superTypes(Integer) AND subTypes(< universe >)
  results in the following new estimation for E([l8]):
      superTypes(Integer)
  Type equivalence related to E([l8]): {E(List7), T(new C()), ...}
      work − list += {E(List7), T(new C()), ...}
```

...

2.3 - Choosing types

Around advice

Since the final estimation for both $E([l_6])$, $E([List_4])$ and $E([List_5])$ ends with *Number*, the types chosen are:

$E([l_6]) = Number$
$E([List_4]) = Number$
$E(List_5) = Number$

...

Main method

Since the final estimation for both $E(List_7)$, $E(List_{10})$, $T(new\ C())$ and $E(Iterator)$ ends with *Double*, the types chosen are:

$E(List_7) = Double$
$E(List_{10}) = Double$
$T(new\ C()) = Double$
$E(Iterator) = Double$

...

2.4 - Find casts to remove

Since the type chosen for $E(Iterator)$ is *Double*, the return of *iter.next*() is also *Double*. Therefore, the cast in expression (*Double*)*iter.next*() is redundant, and then, it is marked to be removed.

Phase 3 - Code Rewrite

```
1       class C<T extends Number> {
2               List1<T> m(List2<T> l3) { return l3; }
3       }
4       aspect A{
5               List4<Number> around(): execution(List m(..)) {
6                       List5<Number> l6 = proceed();
7                       l6.add(1);
8                       return l6;
9               }
10      }
11      class Main {
12              public static void main(String args[]){
13                      List7<Double> l8 = new ArrayList9<Double>();
14                      l8.add(1.1);
15                      List10<Double> l2_11  = new C<Double>().m(l8);
16                      for (Iterator<Double> iter=l2_11.iterator(); iter.hasNext();) {
17                              Double d = ~~(Double)~~iter.next();
18                              ...
19                      }
20              }
21      }
```

REFERENCES

AGESEN, O. The Cartesian Product Algorithm: simple and precise type inference of parametric polymorphism. In: ECOOP '95: PROCEEDINGS OF THE 9TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1995, London, UK. ... Springer-Verlag, 1995. p.2–26.

ASPECTJ. **The AspectJ 5 Development Kit Developer's Notebook**. [S.l.: s.n.], 2005. URL: http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html. Last access: 14/12/2008.

AVGUSTINOV, P. et al. Semantics of static pointcuts in aspectJ. In: POPL '07: PROCEEDINGS OF THE 34TH ANNUAL ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 2007, New York, NY, USA. ... ACM, 2007. p.11–23.

BELLAMY, B. et al. Efficient local type inference. **SIGPLAN Not.**, New York, NY, USA, v.43, n.10, p.475–492, 2008.

BRACHA, G. et al. 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, 1998, New York, NY, USA. ... ACM Press, 1998. p.183–200.

CARDELLI, L.; WEGNER, P. On understanding types, data abstraction, and polymorphism. **ACM Comput. Surv.**, New York, NY, USA, v.17, n.4, p.471–523, 1985.

CRACIUN, F. et al. An Interval-Based Inference of Variant Parametric Types. In: PROGRAMMING LANGUAGES AND SYSTEMS, 18TH EUROPEAN SYMPOSIUM ON PROGRAMMING, ESOP 2009, HELD AS PART OF THE JOINT EUROPEAN CONFERENCES ON THEORY AND PRACTICE OF SOFTWARE, ETAPS 2009, YORK, UK, MARCH 22-29, 2009. PROCEEDINGS, 2009. ... Springer, 2009. p.112–127. (Lecture Notes in Computer Science, v.5502).

DINCKLAGE, D. von; DIWAN, A. Converting Java classes to use generics. In: OOPSLA '04: PROCEEDINGS OF THE 19TH ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 2004, New York, NY, USA. ... ACM Press, 2004. p.1–14.

DONOVAN, A.; ERNST, M. D. **Inference of generic types in Java**. 2003.

DONOVAN, A. et al. Converting java programs to use generic libraries. In: OOPSLA '04: PROCEEDINGS OF THE 19TH ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 2004, New York, NY, USA. ... ACM Press, 2004. p.15–34.

DUGGAN, D. Modular type-based reverse engineering of parameterized types in Java code. In: OOPSLA '99: PROCEEDINGS OF THE 14TH ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 1999, New York, NY, USA. ... ACM Press, 1999. p.97–113.

EIFRIG, J. et al. Sound polymorphic type inference for objects. **SIGPLAN Not.**, New York, NY, USA, v.30, n.10, p.169–184, 1995.

FRAINE, B. D. et al. StrongAspectJ: flexible and safe pointcut/advice bindings. In: AOSD, 2008. ... ACM, 2008. p.60–71.

FUHRER, R. et al. Efficiently refactoring Java applications to use generic libraries. In: ECOOP 2005 — OBJECT-ORIENTED PROGRAMMING, 19TH EUROPEAN CONFERENCE, 2005, Glasgow, Scotland. ... [S.l.: s.n.], 2005. p.71–96.

GAMMA, E. et al. **Design Patterns**. [S.l.]: Addison-Wesley Professional, 1995.

GOSLING, J. et al. **The Java Language Specification Third Edition**. [S.l.]: Addison-Wesley, 2005.

HANNEMANN, J. Aspect-Oriented Refactoring: classification and challenges. In: LATE'06, 5TH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2006. ... [S.l.: s.n.], 2006.

HANNEMANN, J. et al. Refactoring to aspects: an interactive approach. In: PROCEEDINGS OF THE 2003 OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE, 03., 2003, New York, NY, USA. ... ACM, 2003. p.74–78.

IGARASHI, A. et al. Featherweight Java: a minimal core calculus for java and gj. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.23, n.3, p.396–450, 2001.

IGARASHI, A. et al. A Recipe for Raw Types. In: WORKSHOP ON FOUNDATIONS OF OBJECT-ORIENTED LANGUAGES (FOOL), 2001. ... [S.l.: s.n.], 2001.

JAGADEESAN, R. et al. Typed parametric polymorphism for aspects. **Sci. Comput. Program.**, Amsterdam, The Netherlands, The Netherlands, v.63, n.3, p.267–296, 2006.

KICZALES, G. et al. An Overview of AspectJ. In: PROCEEDINGS OF THE EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP'01), 2001. ... Springer, 2001. p.327–353. (Lecture Notes in Computer Science, v.2072).

KIEŻUN, A. et al. Generics-related refactorings in eclipse. In: OOPSLA '05: COMPANION TO THE 20TH ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 2005, New York, NY, USA. ... ACM Press, 2005. p.170–170.

KIEŻUN, A. et al. Refactoring for Parameterizing Java Classes. In: ICSE '07: PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2007, Washington, DC, USA. ... IEEE Computer Society, 2007. p.437–446.

KOPPEN, C.; STÖRZER, M. PCDiff: Attacking the fragile pointcut problem. In: EUROPEAN INTERACTIVE WORKSHOP ON ASPECTS IN SOFTWARE (EIWAS), 2004. ... [S.l.: s.n.], 2004.

MONTEIRO, M. P.; FERNANDES, J. M. Towards a catalog of aspect-oriented refactorings. In: AOSD '05: PROCEEDINGS OF THE 4TH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2005, New York, NY, USA. ... ACM Press, 2005. p.111–122.

MONTEIRO, M. P.; FERNANDES, J. M. L. Towards a catalogue of refactorings and code smells for aspectj. **Transactions on Aspect Oriented Software Development (TAOSD)**, [S.l.], v.Lecture Notes in Computer Science, n.3880, p.214–258, 2006.

MUNSIL, W. Case Study: converting to java 1.5 type-safe collections. **Journal of Object Technology**, [S.l.], v.3, n.8, p.7–14, 2004.

ODERSKY, M.; WADLER, P. Pizza into Java: translating theory into practice. In: POPL '97: PROCEEDINGS OF THE 24TH ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 1997, New York, NY, USA. ... ACM Press, 1997. p.146–159.

PALSBERG, J.; SCHWARTZBACH, M. I. Object-Oriented Type Inference. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA), 1991, New York, NY. **Proceedings...** ACM Press, 1991. v.26, n.11.

PALSBERG, J.; SCHWARTZBACH, M. I. **Object-Oriented Type Systems**. [S.l.]: John Wiley & Sons, 1993.

PLEVYAK, J.; CHIEN, A. A. Precise concrete type inference for object-oriented languages. **SIGPLAN Not.**, New York, NY, USA, v.29, n.10, p.324–340, 1994.

RUBBO, F. B. et al. On the Interaction of Advices and Raw Types in AspectJ. **Journal of Universal Computer Science**, [S.l.], v.14, n.21, p.3534–3555, 2008.

SUTTER, B. D.; DOLBY, J. Customization of Java library classes using type constraints and profile information. In: IN ECOOP, 2004. ... [S.l.: s.n.], 2004. p.585–610.

TIP, F. et al. Refactoring for generalization using type constraints. In: OOPSLA '03: PROCEEDINGS OF THE 18TH ANNUAL ACM SIGPLAN CONFERENCE

ON OBJECT-ORIENTED PROGRAMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 2003, New York, NY, USA. ... ACM, 2003. p.13–26.

TIP, F. et al. **Refactoring techniques for migrating applications to generic Java container classes**. Yorktown Heights, NY, USA: IBM T.J. Watson Research Center, 2004. IBM Research Report. (RC 23238).

TORGERSEN, M. et al. Adding wildcards to the Java programming language. In: SAC '04: PROCEEDINGS OF THE 2004 ACM SYMPOSIUM ON APPLIED COMPUTING, 2004, New York, NY, USA. ... ACM Press, 2004. p.1289–1296.

WLOKA, J. et al. Tool-supported refactoring of aspect-oriented programs. In: AOSD '08: PROCEEDINGS OF THE 7TH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2008, New York, NY, USA. ... ACM, 2008. p.132–143.

YE, L.; VOLDER, K. D. Tool support for understanding and diagnosing pointcut expressions. In: AOSD '08: PROCEEDINGS OF THE 7TH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2008, New York, NY, USA. ... ACM, 2008. p.144–155.