Inference Rules for Generic Code Migration of Aspect-Oriented Programs

Fernando Barden Rubbo\textsuperscript{1}  Eduardo Kessler Piveta\textsuperscript{2}  
Daltro José Nunes\textsuperscript{1}

\textsuperscript{1}Instituto de Informática  
Universidade Federal do Rio Grande do Sul (UFRGS)

\textsuperscript{2}Campus Alegrete  
Universidade Federal do Pampa (UNIPAMPA)

\{fbrubbo,daltro\}@inf.ufrgs.br, piveta@unipampa.edu.br

Abstract. Several changes occurred in the AspectJ language to provide support for parametric polymorphism, a.k.a. generics, which was recently introduced in the Java type system. Such changes aim to improve the type safety of the source code and to prepare the language to support generic code migration. Current approaches for this kind of migration focus only on object-oriented code. Thus, they do not consider the use of aspects to encapsulate crosscutting concerns. We propose a collection of type constraint rules for the polymorphic version of AspectJ. These rules are used together with an existing constraint based algorithm to enable the conversion of non-generic legacy code to add actual type parameters in both Java and AspectJ languages.

1. Introduction

Parametric polymorphism in Java and AspectJ type systems improves the type safety and the expressiveness of source code. To take advantage of this pervasive feature, developers must migrate their code to explicitly supply actual types parameters in both declaration and instantiation sites.

Although those type systems were designed to support that kind of migration, this process – when performed manually – can be tedious, time consuming and error prone (von Dincklage and Diwan 2004; Munsil 2004; Donovan et al. 2004). The reason is that actual type parameters must be inferred to remove as much insecure downcasts as possible without affecting the original semantics of the program. This problem is known as the \textit{instantiation problem} (Donovan et al. 2004).

There are some approaches to help in this migration process (Donovan et al. 2004; von Dincklage and Diwan 2004; Fuhrer et al. 2005). In general, they analyze the 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.

Those 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. Then, approaches focused on the migration in the context of aspect-oriented (AO) languages must deal with the following:

1. **Poor inference**: aspect-oriented constructions must be considered during the program transformation since they can change the inference results. Without analyzing such constructions, the inferred types may not be as good as expected.

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

3. **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 insecure downcasts than the use of an unique type.

4. **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 code (Ye and Volder 2008; Wloka et al. 2008). A simple task of adding actual type parameters to raw declarations and instantiations of generic types, for example, can be troublesome.

This paper presents a novel collection of type constraint rules for the polymorphic version of AspectJ. These rules were used together with the algorithm of Fuhrer et al. (Fuhrer et al. 2005) to address the instantiation problem for both OO and AO contexts, enabling the conversion of non-generic legacy code to add actual type parameters in both Java and AspectJ languages.

Our proposal generates wildcard type constraints and actual type parameters for several OO constructions that were not considered in the original algorithm (Fuhrer et al. 2005). Beyond these improvements – that enable the removal of more insecure downcasts –, our rules also generate type constraints for AO programs. To the best of our knowledge, this is the first work presenting a solution for an automated aspect-aware generic code migration.

Another contribution of this study is the aspect-aware type constraint framework. In this paper, we have used our framework for generic migration of AO code. However, there are other refactorings, such as generalization (Tip et al. 2003) and the customization of container classes (Sutter and Dolby 2004), that can be extended to transform AO programs using our solution.

This paper is organized as follows. Sections 2 and 3 overview the Java and AspectJ generic type systems including their main design decisions. Section 4 presents our inference rules for deriving type constraints and discusses a couple of examples which claim for migration to take advantage of generics. Section 5 describes related works and how our solution improves the current state-of-the-art. Finally, Section 6 includes some concluding remarks.
2. Generic Java

Generic Java (GJ) (Gosling et al. 2005; Odersky and Wadler 1997; Bracha et al. 1998; Igarashi et al. 2001a; Igarashi et al. 2001b) allows the use of generic classes, interfaces and methods to obtain a better compile-time type checking and to improve the readability of the source code – since types help to document the functionalities of programs and the intentions of programmers (Gosling et al. 2005; Donovan et al. 2004).

A generic class (or a generic method) must be declared along with its formal type parameters. The variables bounded to these parameters can be used within the class as normal types in non-static declarations. The following listing shows two different examples of class parametrization and instantiation.

```
1 class C1<F1> { .. }
2 class C2<F2 extends Number, F3> { .. }
3 ..
4 C1<Float> v1 = new C1<Float>();
5 C2<Float, String> v2 = new C2<Float, String>();
```

While C1 (line 1) declares a unique formal type parameter, F1, implicitly bounded to Object, C2 (line 2) declares two type parameters (F2 and F3) bounded to Number and Object, respectively. Whenever a generic class is used (lines 4 and 5), its declarations and instantiations must provide the actual type parameters respecting the bounds defined in the correspondent formal declaration.

2.1. Invariant Sub-Typing

The GJ type system allows a given class C to extends a generic class B. In this kind of inheritance, if there are formal type parameters specified in C, the variables bounded to these parameters can be used as real types during the instantiation of its super class B. Consider the following listing:

```
1 class C4<F2 extends Number> extends C1<F2> { .. }
2 class C5 extends C1<Integer> { .. }
```

In this example, class C4 declares the formal type parameter F2 (bounded to Number) and passes it as the actual type to its super class C1. Similarly, class C5 – which does not have type parameters – always passes Integer to its super class. Note that, unlike arrays, generic types are not co-variants\(^1\), which means that, in a scenario where C4 extends C1, C4\{A1\} is a subtype of C1\{A2\} if and only if A1 is equals to A2. In other words, C4\{Integer\} is not a valid subtype of C1\{Number\} even knowing that Integer is a subtype of Number. As a rule, whenever there is a class C being a subtype of a generic class B, all type parameters defined in this hierarchy must be invariant.

2.2. Raw Type and Type Erasure

A parameterless declaration or instantiation of a generic class is known as a raw type. Raw types were created to ensure the source level backward compatibility (i.e. legacy non-generic code can coexist with the newer polymorphic library versions without modification).

\(^1\)In the current Java specification (Gosling et al. 2005), arrays of generic types (such as, `new ArrayList<String>[0]`) are not allowed.
Although this design provides a convenient support for non-parameterized
generic classes during the generic software evolution (Bracha et al. 1998;
Igarashi et al. 2001b), it forces the type system to have some type rules that are
deliberately unsound (Igarashi et al. 2001b). For example, consider the following
code fragment:

```
1  C4 v2 = new C4<Integer>();
2  C1<Float> v3 = v2;
```

In line 2, an object of the type C4<Integer> is being assigned to a variable
declared as C1<Float>. Since the GJ sub-typing is invariant, this construction would
be an ill-typed statement. However, because of the raw type design, that is a well-
typed construction that will cause a type conversion error at runtime.

Similarly to raw types, the type erasure design aims to solve compatibility is-
isues in the bytecode level. After type checking, the Java compiler erases all generic
information and inserts downcasts in order to create a semantically equivalent byte-
code which is very close to the older and non-generic one. Due to both raw types
and type erasure designs, information about type parameters are not available at
runtime and, consequently, operations such as instanceof are not able to check
those data in the current Java specification (Gosling et al. 2005).

2.3. Wildcards

Wildcards were included into the GJ type system because they augment the expressiveness and
the flexibility of generic declarations (Torgersen et al. 2004;
Gosling et al. 2005). Usually they are useful in situations where only partial knowl-
edge of the type is required. A wildcard can be considered as an existential
type in which it is possible to specify upper and lower bounds (⟨? extends C⟩
and ⟨? super C⟩, respectively). If no bounds are defined, Object is assumed
as the upper bound. Consider the following method declaration as an example,
in which the parameter coll requires a collection of a subclass of Number (i.e.
Collection⟨∃ x . x ≤ Number⟩).

```
void m(Collection<? extends Number> coll) {...}
```

3. Generic AspectJ

AspectJ (Kiczales et al. 2001) is a superset of Java that provides abstractions to
deal with crosscutting concerns, including aspects, inter-type declarations, point-
cuts, advices, and others. It also added support for parametric polymorphism
(AspectJ 2005; Jagadeesan et al. 2006; Rubbo et al. 2008), allowing the use of pa-
rameterized types within aspects, including pointcut expressions, and inter-type declarations.

The polymorphic support provided by AspectJ is an extension of Generic
Java. Thus, due to some design decisions inherited from its foundation (see Section
2.2), the AspectJ type system must deal with the lack of generic information during
the weaving-time – which takes place just after the erasure process.
3.1. Matching Generic Types

The matching mechanism of AspectJ also uses raw types to maintain the source level backward compatibility. This means that when those types are used in pointcut expressions and type patterns, they ensure that pointcuts already written in an existing non-generic code will continue to work as expected when converted to a generic version.

Cooked types\(^2\), though, can be used together with \texttt{args(\ldots)} primitive pointcuts and \texttt{after(\ldots) returning(\ldots)} advices. These constructions work like Java casting conversion\(^3\) with a subtle difference: instead of throwing a ClassCastException when it founds an invalid cast, it does not execute the advice. Consider the following example to illustrate a possible usage for \texttt{args(\ldots)} construction:

```java
public class A\<E extends Number\> {
    void m1(List\<Integer\> e) {} // have a match
    void m2(List\<? extends Number\> e) {} // have an unsafe match
    void m3(List\<Number\> e) {} // does not have a match
}
```

Although the signature matching part (i.e. \texttt{execution(void *(\ldots))}) of the pointcut expression aims to match all methods with a void return type, the primitive \texttt{args(List(Integer))} tells the weaver that only methods with one parameter that can be casted to \texttt{List(Integer)} must have a match. This is why the methods \texttt{m1} and \texttt{m2} have a match and the method \texttt{m3} does not. Note that in the case of \texttt{m2}, the weaver marks the advice with an appropriate warning; since casting \texttt{List(\? extends Number)} to \texttt{List(Integer)} is an insecure conversion.

Additionally, the matching is also affected by type erasure. Since no generic information remains at weaving-time, AspectJ is not able to check which is the actual type of a given type variable. For instance, consider the following declarations:

```java
class C\<E extends Number\> {
    void m1(E e) {} // does not match m1
    void m2(List\<E\> e) {} // matches m2
}
```

The advices declared in line 6 and 8 do not apply to any method because AspectJ does not support matching against type variable references. On the other hand, join points which have this kind of variables – as part of their signature – are matched by their erasure. Therefore, no matter how \texttt{C} is instantiated, the second and the fourth advices (line 7 and 9) are applied to methods \texttt{m1} and \texttt{m2}, respectively, because the erasure of \texttt{E} is \texttt{Number} and the erasure of \texttt{List(E)} is \texttt{List}.

3.2. Inter-type and Parent Declarations

Inter-type and parent declarations can introduce state and behavior to a class, an aspect or an interface. These declarations can adapt class structures without mod-

\(^2\)The term of cooked types is used to refer parameterized types.

\(^3\)See the AspectJ bug \url{https://bugs.eclipse.org/bugs/show_bug.cgi?id=253109}. 

107
ifying 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:

```java
class C {
    void m1(List l) {
        m2(l);
    }
}

aspect A {
    void C.m2(List l) {
        ...
    }
}
```

The class C is well-formed only in the presence of the aspect A, since C depends on A to declare the method m2. Although this kind of definition provides a modular way to organize the code that deals with crosscutting concerns, its use can make it difficult to reason about the behavior of a program. Such constructions also make things difficult to infer the right actual type parameter during the generic code migration.

4. Generic Code Migration

In this section, we extend a model of type constraints used by several authors (Fuhrer et al. 2005; Kieżun et al. 2007; Tip et al. 2003; Sutter and Dolby 2004), which was initially proposed by Palsberg and Schwartzbach (Palsberg and Schwartzbach 1993). Our extended version of the constraints were used together with an existing generic migration algorithm (Fuhrer et al. 2005) to enable the conversion of non-generic legacy code to add actual type parameters in both Java and AspectJ programs.

4.1. Examples

Before describing our aspect-aware generic type constraint framework, we show a couple of examples to demonstrate the complexity added by the use of aspects during generic migration.

Aspect-oriented constructions must be considered during program transformations.

```java
class C1<E extends Number> {
    ...
}
class C2 {
    void m1(List l) {
        ...
    }
}
aspect A {
    declare parents: C2 extends C1;
    void C1.m1(List l) {
        ...
    }
}
```

During the generic migration of the method m1 (line 2), there are three situations that must be considered, otherwise the refactored version of the code will be ill-typed: (i) the declare parents construction (line 4), since it is augmenting the C2 hierarchy; (ii) the inter-type declaration (line 5), since it is adding to C1 a method which is being overridden by the method m1 defined in C2; (iii) since the method m1 is being implicitly overridden in C2, the super-class C1 must be instantiated passing the appropriate type parameters. It is required because every time a subclass of a generic class provides actual type parameters for overriding methods, the super class must be instantiated accordingly (Gosling et al. 2005).

A simple task of adding actual type parameters can be troublesome in aspect-oriented programs.

```java
class C {
    void m1(List l) {...}
    void m2(List l) {...}
}
aspect A {
    before() : execution(void C.*(...))
    & & args(List){ .. }
}
```

108
In this example, adding a type parameter for the `List` declared in method `m1` or `m2` must consider the type declared in the `args(..)` construction. Otherwise, the semantics of the program may change without warnings. This can happen because the advice (declared in the right side of the same example) may stop matching one of the methods. Note that, after the migration, all advices must be applying to exactly the same join points they applied before the refactoring had started.

Wildcards must be considered for aspect definitions.

In cases like our last example, the use of a wildcard is recommended to improve the type safety of the aspect-oriented program. For example, suppose that a developer infers `List<Integer>` for `m1` and `List<Double>` for `m2`; then, the `args(..)` construction must define `List<? extends Number>` to remove the biggest number of insecure downcast.

### 4.2. Basic Concepts and Functions

This section describes the basic notation, functions and concepts needed for the comprehension of the type constraint rules.

<table>
<thead>
<tr>
<th>Notation:</th>
<th>Basic Functions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M ) methods</td>
<td>( \tau ) type of expression/declaration ( E )</td>
</tr>
<tr>
<td>( m ) method names</td>
<td>( \tau_p ) type of ( E ) in the original program</td>
</tr>
<tr>
<td>( F ) fields</td>
<td>( M ) declared type of method return</td>
</tr>
<tr>
<td>( f ) field names</td>
<td>( F ) declared type of field</td>
</tr>
<tr>
<td>( C ) classes, interfaces or aspect</td>
<td>( T(E) ) actual type parameter ( T ) in expression ( E )</td>
</tr>
<tr>
<td>( I ) interfaces</td>
<td>( T(C) ) actual type parameter ( T ) in class ( C )</td>
</tr>
<tr>
<td>( A ) aspects</td>
<td>( Decl(M) ) type that declares method ( M )</td>
</tr>
<tr>
<td>( AD ) advices</td>
<td>( Decl(F) ) type that declares field ( F )</td>
</tr>
<tr>
<td>( T ) formal type parameters</td>
<td>( Param(M, i) ) the ( i )th formal parameter of method ( M )</td>
</tr>
<tr>
<td>( E ) expressions/declarations</td>
<td>( Param(M, e) ) the formal parameter of method ( M ) which is related to ( args(\ldots e\ldots) )</td>
</tr>
</tbody>
</table>
| \( \tau, \alpha \) types       | OnType(M) = \[
|                                | \begin{cases}
|                                | \tau & \text{if } Decl(M) \text{ is Aspect AND } M \text{ is an intertype decl } \tau.m \\
|                                | Decl(M) & \text{otherwise}
|                                | \end{cases}
|                                | OnType(F) = \[
|                                | \begin{cases}
|                                | \tau & \text{if } Decl(F) \text{ is Aspect AND } M \text{ is an intertype decl } \tau.f \\
|                                | Decl(F) & \text{otherwise}
|                                | \end{cases}
|                                | \[
|                                | bound(\tau) = \begin{cases}
|                                | Object & \text{if } \tau \text{ is } T \\
|                                | bound(\tau') & \text{if } \tau = T \text{ extends } \tau' \\
|                                | bound(\tau') & \text{if } ? \text{ extends } \tau' \\
|                                | C & \text{if } \tau = C \\
|                                | C(\tau_1, \ldots, \tau_n) & \text{if } \tau = C(\tau_1, \ldots, \tau_n)
|                                | \end{cases}
|                                | |
|                                | head(C(\tau_1, \ldots)) = C \\
|                                | head(C) = C \\
|                                | | \tau = head(bound(\tau)) |

The left side of the listing shows the basic notation used in this work. Let \( M \) and \( F \) be method and field declarations, respectively, including the complete signature and the reference to the 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 the context. Finally, consider \( \tau \) and \( \alpha \) as any valid AspectJ type (i.e. classes, interfaces, aspects, type variables and wildcards).

The right side of the listing shows a set of basic functions for obtaining information about: the type of an expression ([\( E \)] and [\( E_p \)], the declared type of methods and fields ([\( M \)] and [\( F \)], respectively), the actual type binded to a type variable \( T \) (\( T(E) \) and \( T(C) \)) and the formal parameter declared in a method (\( Param(M, i) \) and \( Param(M, e) \)).
The function returns the erasure of a given type, and the function is responsible to discover which is the declaration type of a method or definition of overriding for AspectJ is the following: A virtual method judgments are provided by rule such coercion is called the.

However, to provide compatibility with old code that instantiates classes without type parameters, raw types are used in place of cooked types even knowing that.

The relation assumes that whenever there is a given type being a subtype of a generic type, then all type parameters defined in this hierarchy must be invariant.

Rule states that the cooked type is a subtype of a raw type OR.

The subtyping relationship is the reflexive and transitive closure (rules s1 and s4, respectively) of the extends/implements relation between types. The s5 relation assumes that whenever there is a given type C being a subtype of a generic type B, then all type parameters defined in this hierarchy must be invariant.

Rule s6 states that the cooked type C(Integer) is a subtype of the raw type C. However, to provide compatibility with old code that instantiates classes without type parameters, raw types are used in place of cooked types even knowing that construction is not safe. The relation that includes safe subtyping but also allows such coercion is called the unsafe subtyping relation. The rules for unsafe type judgments are provided by rule s7.

Since our subtyping relationship understands parent declaration, our definition of overriding for AspectJ is the following: A virtual method M in type overrides a virtual method M1 in type if M and M1 have identical signatures or M is a sub-signature of M1 and τ ≤ 1. This definition allows the creation of the function, which returns the roots, independently if the hierarchy is built by a parent declaration or not.

4Note that for simplicity we are showing in the subtyping relationship only one type parameter, but there can be several type parameters for each class. Please, refer to (Gosling et al. 2005; Igarashi et al. 2001b) for more detailed information about subtyping in Java and in (Rubbo et al. 2008) for subtyping in AspectJ.

5The notion of sub-signature is used to express a relationship between two methods whose signatures are not identical, but in which one override the other (Gosling et al. 2005).
Finally, we present as follows the syntax of type constraints. Each side of the relationship represents constraint variables, which are types associated with program constructs and 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.

\[
\begin{align*}
\tau & = \tau_1 & \text{type } \tau \text{ must be the same as type } \tau_1 \\
\tau & \leq \tau_1 & \text{type } \tau \text{ must be the same as or a subtype of type } \tau_1
\end{align*}
\]

4.3. Aspect-aware Type Constraints

This section presents the rules for deriving type constraint from several Java and AspectJ constructs.

\[
\begin{align*}
P & \text{contains assignment } E_1 = E_2 \\
& \quad \text{Dec1CGen}([E_1, E_2]) \tag{4.1}
\end{align*}
\]

\[
\begin{align*}
P & \text{contains constructor call } E \equiv \text{new } C(E_1, \ldots, E_k) \\
& \quad \text{if } \not\equiv \text{C} \not\equiv \text{not an interface} \tag{4.2}
\end{align*}
\]

\[
\begin{align*}
P & \text{contains call } E_0.\mathtt{m}(E_1, \ldots, E_k) \text{ to virtual method } M \\
& \quad \text{RootDeS}(M) = [\tau_1, \ldots, \tau_k] \tag{4.3}
\end{align*}
\]

\[
\begin{align*}
P & \text{contains cast expression } E \equiv (C)E_0 \\
& \quad \text{Dec1CGen}([E_0], C) \tag{4.4}
\end{align*}
\]

\[
\begin{align*}
P & \text{contains downcast expression } E \equiv (C)E_0 \\
& \quad \text{Dec1CGen}([E_0], C) \text{ is not an interface} \tag{4.5}
\end{align*}
\]

Since the use of wildcards is allowed in declaration sites, rules R1, R5 and R7 (assignment, downcast and method return, respectively) generate constraints using the following function.

\[
\text{Dec1CGen}(\tau_1, \tau_2) = \begin{cases} \\
\{ \tau_1 \leq \tau_2 \} & \text{when } \tau_2 \text{ is not generic type} \quad (dcg1) \\
\{ \text{head}(\tau_1) \leq \text{head}(\tau_2) \} \text{ AND } T_2(\tau_1) = T_2(\tau_2) \text{ OR Wild}(T_1(\tau_1), T_1(\tau_2)) & \text{when } \tau_2 \text{ is generic type} \quad (dcg2)
\end{cases}
\]

Constraints generated by \text{Dec1CGen}(\ldots) must respect the invariant subtyping or the type being assigned (\tau_2) must declare a wildcard. Since the generation of wildcards depends on \tau_1, the function \text{Wild}(\ldots) must be executed only when a fully defined type is estimated for \tau_1.

Rule R3 says that the type of the object calling method \mathtt{m} must be a subtype of the one declaring the method. Rule R4 and R5 are related to casting expression.

\[
\text{CGen}(\alpha \text{ op } C) = \begin{cases} \\
\{ \alpha = C \} & \text{when } \tau \equiv C \quad (rg1) \\
\{ \text{head}(\tau) \equiv \text{head}(C) \} & \text{when } \tau \equiv T(\alpha) \quad (rg2) \\
\{ \alpha = \tau \} \text{ OR Wild}(\alpha, \tau) & \text{when } \tau \equiv \tau \text{ extends } \tau' \quad (rg3) \\
\{ \alpha \equiv \tau \} & \text{when } \tau \equiv \tau' \text{ extends } \alpha \quad (rg4) \\
\{ \alpha \equiv \tau \} \text{ AND CGen}(\tau|\alpha, \tau) & \text{when } C(\tau_1, \ldots, \tau_l) \text{ and } C \text{ is declared as } C(\tau_1, \ldots, \tau_l) \text{ AND } 1 \leq i \leq k \quad (rg5)
\end{cases}
\]

The constraint generation \text{CGen}(\ldots) is the recursive function used to create constraints for generic/non-generic method/constructor calls. It depends on \text{TR}(\ldots) function to resolve \tau – according to its context – before \text{CGen}(\ldots) be called.

\[
\text{TR}(\tau, E_0, E) = \begin{cases} \\
C & \text{when } \tau \equiv C \text{ and } C \text{ is not a generic type} \quad (r1) \\
T(E) & \text{when } \tau = T(E) \text{ and } T(E) \text{ is a type variable declared in the method} \quad (r2) \\
T(E_0) & \text{when } \tau = T(E_0) \text{ and } T(E_0) \text{ has the correspondent actual type for } T_i \quad (r3) \\
1. \text{ extends } T(E_0, E) & \text{when } \tau \equiv \text{extends } T' \text{ and } E_0 \text{ has not the correspondent actual type for } T_i \quad (r4) \\
2. \text{ super } T(E_0, E) & \text{when } \tau \equiv \text{super } T' \quad (r5) \\
C(T_1(E_0, E), \ldots, T_{\ell}(E_0, E)) & \text{when } \tau \equiv C(T_1, \ldots, T_{\ell}) \quad (r7) \\
C(T_1(\tau_1, \ldots, \tau_l)) & \text{when } \tau \equiv C \text{ and } C \text{ is a generic type declared as } C(T_1, \ldots, T_{\ell}) \quad (r8)
\end{cases}
\]
Rules R8 and R9 are used to generate constraints for method calls. For each method call, these rules create a set of constraints related to method returning and parameters.

Rule R10 complements rule R2 which refers to constructors calls (allocation sites). This rule is needed because, similarly to methods, constructors can also have generic types in their parameters. Note that, since we can not declare type variable neither in constructors nor in fields; rules R10, R11 and R12 are passing a null value, represented by a hole (•), in the third parameter of TR(•). Rule R13 takes care of subtype relationship among generic library classes. This rule propagates actual type parameters from a given type to its super-type.

The constraint generation above, \textit{OverCGen}(\ldots), is the auxiliary function used to recursively create constraints for method overriding, independently if the super class is generic or not.

While rules R14 and R15 address the constraint generation for method overriding, rules R16 and R17 were created to manage method calls declared in the super class. Those rules are required because GJ allows library designers to freely generate methods independently of clients that define subclasses or sub-interfaces of the library (Gosling et al. 2005). Moreover, every time a subclass of a generic class provides actual type parameters for overriding methods, the super class must be instantiated passing the appropriate type parameters. Otherwise, the hierarchy is ill-typed (Gosling et al. 2005). This implies that all constraint generated by rules R14, R15, R16 and R17 must be discarded if no type constraint that adds actual type parameters for the super class be found.

Since all type rules presented so far are aware of inter-type and parent declarations, we should still define rules for advised methods\footnote{We are assuming that the weaver is able to provide all matching information. For example, it can answer the following questions: the advice \texttt{AD} matches the method \texttt{C.m}?, the method \texttt{C.m} is advised by which advices?}. For that, the \texttt{args(\ldots)}
pointcut primitive, after() returning(...) and around advices must be considered since cooked types can be used along with those constructions.

Since AspectJ is not able to match against type variables, rule R18 is used to match join points with this kind of variables by their erasure. It is important to highlight that if the args parameter is a subtype of the method’s parameter, matching happens but it requires a runtime verification. As we already know, type parameters can not be verified at runtime, then a warning is raised for these cases.

Rules R19 and R20, though, say that the types must respect the invariant subtyping or the type declared as an args parameter must use a wildcard. While R19 works for fully defined types, R20 works for parameterized types with type variables.

In summary, rules R21 and R22 say that the around advice returning must be Object or it must be assignable to the statically resolved method return. The function STR(…) is required in rule R22 since every type variable must be replace by its bound before the constraint is generated.

4.4. Discussion
This section shows the results of the automated generic migration proposed by (Fuhrer et al. 2005) when our version of the rules are used. The general idea of the algorithm is divided in three phases:

1. Generation of Constraints: type constraints are generated in the first part of the algorithm, in which each node of the abstract syntax tree (AST) is visited. The output of this phase is a non-empty set of type constraints for each program construction that express the relationships between the declared types of the expressions comprising each construct.

2. Constraints Solving: the type constraints created in the first phase are iterated until a set of legal types for each constraint variable be computed. Since there are usually more than one legal type associated with each constraint variable, the most specific type is chosen in each iteration. At the end of this process, the solver yields an unique type for each constraint variable.

We have not shown in this paper rules for after() returning(...) due to space limitation.
3. **Code Rewrite**: this phase rewrites allocation sites and declarations that refer to generic types. Any redundant cast in the resulting program is removed.

Our set of rules are used in the first phase of the algorithm to define which are the type constraints for a given program. It is important to highlight that the Wild(…) function is applied in the constraints generation phase to make it clear where wildcards may be inferred. However, this function must be executed only in the constraints solver phase whenever the most specific type is chosen.

Applying the algorithm described above in the examples shown in Section 4.1, we have the following:

```java
1 class C1<E extends Number>{...}
2 class C2 { void m1(List<Integer> l){} }
3 aspect A{
4     declare parents:C2 extends C1<Integer>;
5     void C1.m1(List<E> l){}
6 }
7 ...
8 new C2().m1(listOfInteger);
```

Since a list of integers (line 8) is being passed as a parameter to method m1 (line 2), the value inferred by the algorithm is List⟨Integer⟩. Note that this inference implies that the implicitly declared super class, C1, be instantiated also with Integer (line 4). This is required to make the inheritance well-typed. If no actual type parameter were passed to C1, the List parameter declared in line 2 should remain raw.

```java
1 class C{
2     void m1(List<Integer> l){}
3     void m2(List<Double> l){}
4 }
5 aspect A{
6     before(): execution(void C.m*(..))
7         && args(List<? extends Number>){..}
8 }
9 ...
10 new C().m1(listOfInteger);
11 new C().m2(listOfDouble);
```

In this example, since a list of integers and a list of doubles (lines 10 and 11) are being passed as parameters to methods m1 and m2 (lines 2 and 3), respectively; the algorithm infers List⟨Integer⟩ for m1 and List⟨Double⟩ for m2. Note that the List declared in args(…) construction is using a wildcard (line 7). This inference allows the method declarations to define types as specific as possible, removing then, a bigger number of insecure downcast when compared with a non-wildcard inference.

Beyond the support for AO programs, our solution also generates type constraints for the following OO constructions that were not considered in previous works: we accommodate sub-classes of generic classes\(^8\); we generate wildcard types when applicable; we also accommodate raw types used in signature of methods declared into generic classes and; calls for methods defined in the super generic class.

\(^8\)Although Fuhrer et al. (Fuhrer et al. 2005) claims that they accommodate sub-classes of generic classes, they do not show the type rules neither in the paper nor in the implementation. Moreover, they do not mention the cases where it is not possible to infer type parameter in the sub-class due to GJ limitations.
5. Related Work

There are several solutions (Donovan et al. 2004; Fuhrer et al. 2005; von Dincklage and Diwan 2004) for migrating the clients of generic classes to inform actual type parameters. Since these approaches have been proposed for migrating Java programs, they are not able to ensure that the migration will be successful in the presence of aspects. Our proposal, though, mainly differs from these approaches in the sense that we enable the conversion of non-generic legacy code to add actual type parameters in both Java and AspectJ languages.

Donovan et al. (Donovan et al. 2004) proposed an algorithm based on type rules to rewrite clients of generic classes to inform actual type arguments. They tried to ensure that the inferred types are as specific as possible in an effort to minimize the number of insecure downcasts needed. Fuhrer et al. (Fuhrer et al. 2005) also presented a solution based on type constraints. Their proposal focus on efficiency and accuracy and it mainly differs from (Donovan et al. 2004) in the sense that their algorithm is more scalable and it is incorporated into a popular integrated development environment (i.e. Eclipse\(^9\)).

We extend the Fuhrer et al.'s algorithm (Fuhrer et al. 2005) by providing the generation of type constraints for: (i) aspect-oriented constructions; (ii) subclasses of generic classes; (iii) wildcard types; (iv) raw types used in the signature of methods declared into generic classes and; (v) calls for methods defined in the super generic class.

6. Conclusion

In this paper, we have presented a set of type constraint rules which are used together with a generic migration algorithm. This algorithm, when using our type constraints, addresses the instantiation problem in both OO and AO contexts, enabling the conversion of a non-generic legacy code to add actual type parameters in both Java and AspectJ languages.

Our set of rules improves current proposals since it generates type constraints for: AO programs, wildcard types and actual type parameters for several OO constructions that were not considered by related works. Thus, beyond the support for polymorphic AspectJ, these improvements also enable our version of the generic migration algorithm to remove more insecure downcasts than the original proposal. This offer significant benefits since legacy non-generic aspects and classes can take advantage of compiler-enforcement type-safe.

Future work will focus on using our aspect-aware generic type constraint framework to extend other refactorings to consider aspects. Those refactorings are: generalization, parameterization of classes and the customization of container classes.

References


\(^9\)www.eclipse.org


