Supporting covariant return types and generics in type relaxed weaving

Tomoyuki Aotani
Japan Advanced Institute of Science and Technology
aotani@jaist.ac.jp

Manabu Toyama
University of Tokyo
touyama@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
University of Tokyo
masuhara@acm.org

ABSTRACT

This paper introduces our ongoing study on type safety of the type relaxed weaving mechanism in the presence of two Java 5 features, namely covariant return types and generics. We point out additional conditions that are necessary to ensure type safety, which can be checked by a slightly modified type checking rules for the type relaxed weaving.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Aspect-oriented programming, Type relaxed weaving, Covariant return types

1. INTRODUCTION

The around advice is one of the unique and important features in the aspect-oriented programming (AOP) languages based on the pointcut and advice mechanism [9] such as AspectJ [4,7]. It allows us to change the receiver and argument values of method/constructor calls, and also to replace operations with others without modifying the source code of the program. There has been several studies that address improving generality and/or applicability of around advice [3, 8], as well as those design a formal calculus and study type safety for AOP languages with around advice [1, 2, 6].

The type relaxed weaving [8] is a bytecode-level weaving mechanism for AspectJ family of languages. It improves applicability of around advice. It allows a piece of around advice to have a different return type from those of the join points where it is woven. We call such advice *type-relaxing advice* in this paper. Type safety of the type relaxed weaving is proved formally based on an object-oriented calculus called Featherweight Java for Relaxation (FJR) [8], which is an extension to Featherweight Java [5].

This paper introduces our ongoing study on type safety of the type relaxed weaving in the presence of advanced language features that FJR does not have, especially, covariant return types and generics. The covariant return type extension allows a class to override a method with a return type smaller (more specific) than that of the method in its superclass. The generics feature enables us to define generic classes and methods through type parametrization.

The contributions of the paper are as follows:

- We point out additional conditions that are necessary to ensure type safety of the type relaxed weaving in the presence of covariant return types and generics.
- We show that a small modification to the constraint generation algorithm for the original type relaxed weaving is sufficient to support covariant return types.

The rest of the paper is organized as follows. We first visit the type relaxed weaving and see the conditions that advice should satisfy in Section 2. Section 3 shows the problem to support covariant return types and generics. Section 4 explains the basic idea of our solution along with our type-checking algorithm. Section 6 concludes the paper.

2. TYPE RELAXED WEAVING

The type relaxed weaving [8] is a type-safe bytecode-level weaving mechanism for AspectJ. It allows around advice to have a different return type from the join point shadows' on which it is woven. More specifically, it relaxes the typing rule in AspectJ that restricts the return type of a piece of around advice to either the return type of its target join points or one of its subtypes.

Figure 1 is a simple example allowed by the type relaxed weaving, but not allowed by AspectJ. Lines 1–9 are skeletons of three Java classes `Object`, `Integer` and `BigInteger`. Lines 11–18 define the interface `Stream` and two classes `IntegerStream` and `BigIntegerStream`. Lines 20–26 defines a method that creates a `BigIntegerStream` object, picks up a `BigInteger` object from it and converts it into a `String` object. The boxed string at line 23 is the signature of the method call. Finally lines 29–31 define a piece of around advice that replaces a `BigIntegerStream` object with an `IntegerStream` object upon creation.

If we ignore the static types of local variables, it is safe to replace the expression `new BigIntegerStream()` at line

FOAL'11, March 21, 2011, Pernambuco, Brazil.

```
1  // Skeletons of Java classes
2  class Object{ String toString(){...} }
3  class Integer extends Object{
4    String toString(){...}
5  }
6  class BigInteger extends Object{
7    String toString(){...}
8    BigInteger abs(){...}
9  }

11 // Definitions of stream classes
12 interface Stream{ Object get(); }
13 class IntegerStream implements Stream{
14   Object get(){...}
15 }
16 class BigIntegerStream implements Stream{
17   Object get(){...}
18 }

20 //in a class
21 void m(){
22   BigIntegerStream bs = new BigIntegerStream();
23   Object o = bs.get(); [Object BigIntegerStream.get()]
24   String s = o.toString();
25   /* s is never used below */
26 }

28 //in an aspect
29 IntegerStream around():call(BigIntegerStream.new()){
30   return new IntegerStream();
31 }
```

Figure 1: `Streams` and type-relaxing advice

22 with `new IntegerStream()`. This is because each of the classes is a subtype of `Stream` and the resulting object is used only as a `Stream` object. Note that the assumption is reasonable since local variables have no type information at bytecode-level in Java.

Intuitively, the type relaxed weaving checks such conditions in a Java bytecode program. Given a piece of around advice `a` and a join point `jp` where `a` is applied to, it checks consistency between the return type of `a` and the operations that use the return value from `jp`. The usages are a method (or constructor) call parameter, a method call target, a return value from a method, a field access target, an assigned value to a field, an array access target and an exception to throw.

In the example, the return type of the advice is `IntegerStream`. The join point is `new BigIntegerStream()` at line 22. The return value from the join point is used as the target of a method call (line 23) whose signature is `Object BigIntegerStream.get()`.

We can safely change the receiver's type in the signature to `Stream` because the former overrides the latter and it does not change the behavior of the program with respect to the semantics of `invokevirtual`/`invokeinterface`. For the same reason, we can safely invoke `Stream.get()` on an `IntegerStream` object.

3. PROBLEM

```
1  // Redefining IntegerStream and BigIntegerStream
2  class IntegerStream{
3    Integer get(){...} //refining return type
4  }
5  class BigIntegerStream{
6    BigIntegerStream get(){...} //refining return type
7  }
8  //in a class
9  void m(){
10   BigIntegerStream s = new BigIntegerStream();
11   BigInteger i = s.get(); [BigInteger BigIntegerStream.get()]
12   BigInteger absi = i.abs();
13   /* s is never used */
14 }
15 /* and the same aspect*/
```

Figure 2: An example using covariant return types that goes type-unsafe after the advice in Figure 1 is woven

The type relaxed weaving is based on the Java 1.4 language, which lacks recent features covariant return types and generics. Supporting those features in type relaxed weaving is not straightforward as we discuss below.

3.1 Support for covariant return types

It is not enough to care about the usage of the return value from a join point when a class can override its superclass's method with a smaller return type. In this section we assume that the language that employs the type relaxed weaving, i.e., RelaxAJ [8], is slightly extended so that it can accept covariance of the return type of a method in the base code.

Figure 2 is a part of the program modified from Figure 1 in which `IntegerStream` and `BigIntegerStream` override the method `get` with the smaller return types, namely `Integer` and `BigInteger` in `IntegerStream` and `BigIntegerStream`, respectively. The method `m` is also changed so that it calls `abs` defined in `BigInteger`.

The around advice shown in Figure 1 can still be woven on the join point shadow `new BigIntegerStream()` at line 10 in Figure 2 because the return value from the join point is only used as the receiver object to invoke `BigIntegerStream.get()`, which overrides `Stream.get()`. Again invoking `Stream.get()` on `IntegerStream` is safe and thus the condition of the type relaxed weaving is satisfied.

The woven code is, however, no longer type-safe. In fact, invoking `abs` at line 12 fails because the receiver `s` is now an `Integer` object, which is the return value of `get` invoked on the return value of the around advice, that is, `IntegerStream`.

3.2 Support for generics

Relaxing return types with type parameters has the same problem to the covariant return types case.

Figure 3 shows an example from which the type relaxed weaving would generate type-unsafe code by applying type-relaxing advice. Lines 3–7 define a generic `Stream` class, which is intended to be used instead of the `Stream` class and the classes implementing it defined in the previous exam-

```
1  ...
2  // Stream class using generics
3  class Stream<X>{
4    X x;
5    Stream(X x){ this.x=x; }
6    X get(){...}
7  }
8  ...
9  //in some class
10 void m(){
11   Stream<BigInteger> s =
12     new Stream<BigInteger>(new BigInteger("0"));
13   BigInteger i = s.get();          Object Stream.get()
14   BigInteger absi = i.abs();  BigInteger BigInteger.abs()
15   /* s is never used below */
16 }

18 //in some aspect
19 Stream<Integer> around():
20   call(Stream.new(Object))&&!within(/*the aspect*/){
21   return new Stream<Integer>(new Integer("0"));
22 }
```

Figure 3: An example using generics that goes type-unsafe after the advice is woven

ples. The method `m` is also modified so that it now uses the generic `Stream` class instead of `BigIntegerStream`. It invokes the method `get` at line 13, whose signature is `Object Stream.get()`.

Here we cannot know with which type the type parameter `X` of `Stream<X>` should be replaced because such information is erased in Java bytecode. The around advice is modified similarly (lines 19–22), i.e., its pointcut specifies creations of `Stream` objects and it returns a `Stream<Int>` object.

Because the pointcut matches `new Stream` at line 12 and `Object Stream.get()` can be invoked on a `Stream<Int>` object, it is allowed to weave the advice on the shadow.

The generated code is again no longer type-safe; it has an invocation to `abs` on `Integer`, which always fails.

4. OUR APPROACH

This section first shows the basic idea of our solution to the problems, and then gives the algorithm $\mathcal{G}^{\mathtt{C}}$ for Featherweight Java for Relaxation with covariant return types ($\mathrm{FJR}^{\mathtt{C}}$) to generate subtyping constraints from a given expression and a type environment. The algorithm is a small extension to the constraint generation algorithm $\mathcal{G}$ for FJR. Although our algorithm does not support generics, it would be easy if the base calculus $\mathrm{FJR}^{\mathtt{C}}$ is extended to support it.

4.1 Basic idea

The basic idea of our solution is to extend the consistency checking rules so that they check the usage of the return values not only from the target join point but also the method calls that uses values *derived* from it. The definition of derived values is given below. If any inconsistencies are found, the advice is rejected.

Let `v` and `w` be values. We say that a method `m` *directly derives* `v` from `w` if `w` is the return value from a method call `v.m`. We also say that `w` is *derived* from `v` if

$$\mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{x}) = (\emptyset, \Gamma(\mathtt{x}))$$

$$\mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{let\ x\ =\ e_1\ in\ e_2}) =$$
$$\quad \textbf{let } (\mathcal{R}_1^{\mathtt{C}}, \mathtt{U}_1) = \mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{e}_1) \textbf{ in}$$
$$\quad \textbf{let } (\mathcal{R}_2^{\mathtt{C}}, \mathtt{U}_2) = \mathcal{G}^{\mathtt{C}}((\Gamma, \mathtt{x}{:}\mathtt{U}_1), \mathtt{e}_2) \textbf{ in}$$
$$\quad (\mathcal{R}_1^{\mathtt{C}} \cup \mathcal{R}_2^{\mathtt{C}}, \mathtt{U}_2)$$

$$\mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{e}_0\mathtt{.m(e}_1\mathtt{,}\ \cdots\ \mathtt{,e}_n\mathtt{)}) =$$
$$\quad \textbf{let } (\mathcal{R}_0^{\mathtt{C}}, \mathtt{U}_0) = \mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{e}_0) \textbf{ in}$$
$$\quad \textbf{let } (\mathcal{R}_1^{\mathtt{C}}, \mathtt{U}_1) = \mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{e}_1) \textbf{ in}$$
$$\quad\quad \vdots$$
$$\quad \textbf{let } (\mathcal{R}_n^{\mathtt{C}}, \mathtt{U}_n) = \mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{e}_n) \textbf{ in}$$
$$\quad \textbf{let } \overline{\mathtt{T}} \to \mathtt{T} = \mathit{mtype}(\mathtt{m}, \mathit{typeOf}(\mathtt{e}_0)) \textbf{ in}$$
$$\quad \textbf{let } \mathtt{V} = \bigcup \mathit{mdeftypes}(\mathtt{m}, \mathit{typeOf}(\mathtt{e}_0)) \textbf{ in}$$
$$\quad (\mathcal{R}_0^{\mathtt{C}} \cup \mathcal{R}_1^{\mathtt{C}} \cup \cdots \cup \mathcal{R}_n^{\mathtt{C}} \cup \{\overline{\mathtt{U}} <: \overline{\mathtt{T}}\}$$
$$\quad\quad \cup \{\mathtt{U}_0 <: \mathtt{X}, \mathtt{X} <: \mathtt{V}, \lambda s.\mathit{mrtype}(\mathtt{m}, s\mathtt{X}) <: \mathtt{Y}\},$$
$$\quad \mathtt{Y})$$
$$\quad \text{(for fresh } \mathtt{X} \text{ and } \mathtt{Y})$$

$$\mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{new\ C()}) = (\emptyset, \mathtt{C})$$

$$\mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{(?e_1 : e_2)}) =$$
$$\quad \textbf{let } (\mathcal{R}_1^{\mathtt{C}}, \mathtt{U}_1) = \mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{e}_1) \textbf{ in}$$
$$\quad \textbf{let } (\mathcal{R}_2^{\mathtt{C}}, \mathtt{U}_2) = \mathcal{G}^{\mathtt{C}}(\Gamma, \mathtt{e}_2) \textbf{ in}$$
$$\quad (\mathcal{R}_1^{\mathtt{C}} \cup \mathcal{R}_2^{\mathtt{C}}, \mathtt{U}_1 \cup \mathtt{U}_2)$$

Figure 4: Modified constraint generation algorithm

- some method `m` directly derives `w` from `v`, or
- `w` is derived from `v'` and `v'` is derived from `v` for some value `v'`.

In Figure 2, the return value from the join point `new BigIntegerStream()` at line 12 is assigned to `s`. We use the variable names to denote the return values for simplicity. `i` and `absi` are derived from `s` because `i` is directly derived from `s` and `absi` is directly derived from `i`.

Our extended rules check whether the return type of the method that directly derives `i` (resp. `absi`) and the operations that use `i` (resp. `absi`) are consistent if `s` is an `IntegerStream` object. The operation `s.get()` directly derives `i` and its return type is `Object`. `i.abs()` at line 12 uses `i` and its signature is `BigInteger BigInteger.abs()`, which can be no more relaxed and inconsistent with `Object`. Hence the rules rejects the around advice (lines 29–31 in Figure 1).

4.2 Constraint generation algorithm

We design an algorithm for the extended consistency checking rules on a small object-oriented language called Featherweight Java for Relaxation with covariant return types ($\mathrm{FJR}^{\mathtt{C}}$), which is a simple extension to FJR [8]. In this section we first give the syntax rules of $\mathrm{FJR}^{\mathtt{C}}$, which is the same to the ones of FJR, and the typing rules that are modified to support covariant return types. Then we give the algorithm $\mathcal{G}^{\mathtt{C}}$ and explain it through a simple example. Proving its formal correctness is not completed yet, which is left for our future work.

The syntax rules of FJR$^{\mathtt{C}}$ are the same to the ones of FJR:

$$\begin{array}{rcl} \mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C\ implements\ }\overline{\mathtt{I}}\mathtt{\ \{\ }\overline{\mathtt{M}}\mathtt{\ \}} \\ \mathtt{M} & ::= & \mathtt{T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{\ return\ e;\ \}} \\ \mathtt{IF} & ::= & \mathtt{interface\ I\ \{\ }\overline{\mathcal{S}}\mathtt{\ \}} \\ \mathcal{S} & ::= & \mathtt{T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{);} \\ \mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{new\ C()} \\ & & \mid \mathtt{let\ x\ =\ e\ in\ e} \mid \mathtt{(?e:e)} \\ \mathtt{S,T} & ::= & \mathtt{C} \mid \mathtt{I} \\ \mathtt{U,V} & ::= & \mathtt{T} \mid \mathtt{U} \cup \mathtt{U} \end{array}$$

An overline denotes a sequence, e.g., $\overline{\mathtt{x}}$ is shorthand for $\mathtt{x}_1,\ldots,\mathtt{x}_n$. The metavariable `C` ranges over class names; `I` ranges over interface names; `m` ranges over method names; and `x` and `y` range over variables, which include the special variable `this`.

`CL` is a class declaration, consisting of its name, a superclass name, interface names that it implements, and methods $\overline{\mathtt{M}}$; `IF` is an interface declaration, consisting of its name and method headers $\overline{\mathcal{S}}$.

The syntax of expressions includes `let` expressions to illustrate the cases when a value returned from around advice is used as values of different types. `let` is the only binding construct of an expression and the variable `x` in $\mathtt{let\ x\ =\ e_1\ in\ e_2}$ is bound in $\mathtt{e_2}$. It also includes non-deterministic choice `(?e:e)` to handle the cases when a variable contains values of different types.

`S` and `T` stand for simple types, i.e., class and interface names, and will be used for types written down in classes and interfaces. `U` and `V` stand for union types. For example, a local variable of type $\mathtt{C} \cup \mathtt{D}$ may point to either an object of class `C` or that of `D`.

To support covariant return types, we need to change the typing rule T-Class, the predicate *override* and the constraint generation algorithm $\mathcal{G}$ to allow each overriding method to have a return type that is a subtype of the one of the method in its superclass and interfaces.

The modified typing rule T-Class is given as follows:

$$\frac{\forall \mathtt{m}, \mathtt{I} \in \overline{\mathtt{I}}.\ \left\{ \begin{array}{l} (\mathit{mtype}(\mathtt{m},\mathtt{I}) = \overline{\mathtt{T}} \to \mathtt{T}_0) \Longrightarrow (\mathit{mtype}^C(\mathtt{m},\mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{S}_0) \\ \text{and } \mathtt{S}_0 <: \mathtt{T}_0 \end{array} \right\} \quad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}}{\mathtt{class\ C\ extends\ D\ implements\ }\overline{\mathtt{I}}\mathtt{\ \{\ }\overline{\mathtt{M}}\mathtt{\ \}\ OK}}\ (\text{T-Class})$$

where $\mathit{mtype}(\mathtt{m},\mathtt{I})$ and $\mathit{mtype}^C(\mathtt{m},\mathtt{C})$ are the functions that return It defines `C` is well-typed if all methods are well typed and all methods declared in $\overline{\mathtt{I}}$ are implemented in `C` with signature that has a smaller return type.

The predicate $\mathit{override}(\mathtt{m},\mathtt{C},\overline{\mathtt{T}}\to\mathtt{T}_0)$, which checks whether `m` is correctly overrides the method of the same name in `C`, is modified similarly as follows:

$$\frac{(\mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{S}} \to \mathtt{S}_0) \Longrightarrow \overline{\mathtt{S}} = \overline{\mathtt{T}} \text{ and } \mathtt{S}_0 <: \mathtt{T}_0}{\mathit{override}(\mathtt{m},\mathtt{C},\overline{\mathtt{T}}\to\mathtt{T}_0)}$$

The modified constraint generation algorithm $\mathcal{G}^{\mathtt{C}}$ (Figure 4) takes a type environment Γ and an expression `e` and returns a set $\mathcal{R}^{\mathtt{C}}$ of extended subtyping constraints and a type `U`. An extended subtyping constraint is an inequality of the form $\mathtt{U} <: \mathtt{V}$ or $\lambda s.\mathit{mrtype}(\mathtt{m}, s\mathtt{X}) <: \mathtt{U}$ where `U` and `V` range over either simple types or variables `X` and `Y`, $\mathit{mrtype}(\mathtt{m},\mathtt{T})$ returns the return type of the method `m` in the simple type `T`, and $\lambda s.\mathit{mrtype}(\mathtt{m}, s\mathtt{X})$ is a function that takes a substitution S of simple types $\overline{\mathtt{T}}$ for type variables $\overline{\mathtt{X}}$ and returns a simple type $\mathit{mrtype}(\mathtt{m}, S\mathtt{X})$. $\mathit{typeOf}(\mathtt{e})$ denotes the simple type of a receiver `e` of a method invocation. $\mathit{mdeftypes}(\mathtt{m},\mathtt{T})$ collects the set of `T`'s supertypes that define `m`.

The case for method invocations is different from the original constraint generation algorithm $\mathcal{G}$. The type variable `X` stands for the receiver type, which has to be a supertype of the expression $\mathtt{e}_0$. $\mathtt{X} <: \mathtt{V}$ guarantees that the receiver type has method `m` whose argument types are $\overline{\mathtt{T}}$. The type variable `Y` stands for the return type, which depends on the receiver type. $\lambda s.\mathit{mrtype}(\mathtt{m}, s\mathtt{X}) <: \mathtt{Y}$ represents this fact.

```
1 Object m(){
2   return
3     let s = (?new BigIntegerStream():
4                new IntegerStream())
5     in let i = s.get()  BigInteger BigIntegerStream.get()
6     in let absi = i.abs()  BigInteger BigInteger.abs()
7     in new Object();
8 }
```

Figure 5: An example code of FJR$^{\mathtt{C}}$

Example.

The method `m` in Figure 2 can be written in FJR$^{\mathtt{C}}$ as Figure 5. The return type is changed from `void` to `Object` because FJR$^{\mathtt{C}}$ does not have it. The around advice is woven manually to lines 3–4 by using non-deterministic choice.

Our algorithm $\mathcal{G}^{\mathtt{C}}$ correctly rejects the program as follows. Applying $\mathcal{G}^{\mathtt{C}}$ to lines 3–4, we get to know that the type of `s` is $\mathtt{BigInteger} \cup \mathtt{Integer}$. At line 5, $\mathcal{G}^{\mathtt{C}}$ generates the constraints:

$$\left\{ \begin{array}{c} \mathtt{BigIntegerStream} \cup \mathtt{IntegerStream} <: \mathtt{X}_1, \\ \mathtt{X}_1 <: \mathtt{BigIntegerStream} \cup \mathtt{Stream}, \\ \lambda s.\mathit{mrtype}(\mathtt{get}, s\mathtt{X}) <: \mathtt{Y}_1 \end{array} \right\}$$

Because $\mathtt{BigIntegerStream} \cup \mathtt{Stream}$ can be reduced to `Stream`, `Stream` is the only candidate for $\mathtt{X}_1$. We can also reduce the constraint for $\mathtt{Y}_1$ and get $\mathtt{Object} <: \mathtt{Y}_1$, which indicates that $\mathtt{Y}_1$ must be `Object`.

Evaluating $\mathcal{G}^{\mathtt{C}}$ on `i.abs()` at line 6, we get the constraint:

$$\left\{ \mathtt{Y}_1 <: \mathtt{X}_2,\quad \mathtt{X}_2 <: \mathtt{BigInteger},\quad \lambda s.\mathit{mrtype}(\mathtt{abs}, s\mathtt{X}_2) <: \mathtt{Y}_2 \right\}$$

It is not satisfiable because there is no simple type such that $\mathtt{Object} <: \mathtt{X}_2 <: \mathtt{BigInteger}$.

5. RELATED WORK

Featherweight Aspect GJ (FAGJ) [6] is a small calculus based on Featherweight GJ [5], which supports covariant return types and generics. Its focus is on studying the incorporation of generic types in AspectJ family of languages, i.e., the authors discuss about typeability and type safety of aspect-oriented programs with generics for the case when the information about type parameters is not available (as in Java bytecode) as well as when it is (as in source code).

StrongAspectJ [3] is another calculus based on Featherweight Java, which focuses on improving generality of advice in a type-safe manner. It supports covariant return types and generics, as FAGJ.

Our study can be seen as a first attempt to connect these work and the type relaxed weaving.

6. CONCLUSIONS AND FUTURE WORK

In order to support the Java 5 features such as the covariant return types and the generics, the type relaxed weaving should be extended to check, in addition to the type usages of the return value from a target join point, those of the derived values from the return value. The additional checks are straightforwardly incorporated into the type relaxed weaving by slightly modifying a rule for overriding a method.

We also gave an extended constraint generation algorithm for Featherweight Java for Relaxation with covariant return types. Although the paper does not include the formalization of the generics support, we presume that no special extension is needed with respect to the consistency checks of the derived values.

As well as proving the correctness of our constraint generation algorithm and implementing the compiler, generics is left for our future work. An interesting technical challenge is to find a type parameters from an object creation expression in Java bytecode, which employ the type-erasure strategy for generics.

7. REFERENCES

[1] Curtis Clifton and Gary T. Leavens. MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63(3):321–374, 2006.

[2] Bruno De Fraine, Erik Ernst, and Mario Südholt. Essential AOP: the A calculus. In *Proceedings of ECOOP'10*, pages 101–125, 2010.

[3] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. StrongAspectJ: Flexible and safe pointcut/advice bindings. In *Proceedings of AOSD'08*, pages 60–71, 2008.

[4] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of AOSD'04*, pages 26–35, 2004.

[5] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

[6] Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63(3):267–296, 2006.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP'01*, pages 327–353, 2001.

[8] Hidehiko Masuhara, Atsushi Igarashi, and Manabu Toyama. Type relaxed weaving. In *Proceedings of AOSD'10*, pages 121–132, 2010.

[9] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of ECOOP'03*, pages 2–28, 2003.