Type Relaxed Weaving*

Hidehiko Masuhara[1], Atsushi Igarashi[2], and Manabu Toyama[1]

[1] Graduate School of Arts and Sciences, University of Tokyo
{masuhara,touyama}@graco.c.u-tokyo.ac.jp
[2] Graduate School of Informatics, Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Abstract. Statically typed aspect-oriented programming languages restrict application of around advice only to the join points that have conforming types. Though the restriction guarantees type safety, it can prohibit application of advice that is useful, yet does not cause runtime type errors. To this problem, we present a novel weaving mechanism called *type relaxed weaving*, that allows such advice applications while preserving type safety. We formalized the mechanism, and implemented as an AspectJ compatible compiler called *RelaxAJ*.

1 Introduction

The advice mechanism is a powerful means of modifying behavior of a base program without changing its original text. AspectJ[11] is one of the most widely used aspect-oriented programming (AOP) languages that support the advice mechanism. It is, in conjunction with the mechanism called the inter-type declarations, shown to be useful for modularizing crosscutting concerns, such as logging, profiling, persistency and security enforcement[2, 4, 18, 21].

One of the unique features of the advice mechanism is the *around advice*, which can change parameters to and return values from operations, or *join points*, in program execution. With around advice, it becomes possible to define such aspects that directly modify values passed in the program, for example caching results, pooling resources and encrypting parameters. Around advice can also modify a system's functionalities by inserting proxies and wrappers to objects that implement the functionalities, or by replacing the objects with new ones.

Statically typed AOP languages restrict pieces of around advice so that they will not cause type errors. Basically, given a piece of around advice that replaces a value with a new one, those languages permit the advice if the type of the new value is the same type or a subtype of the join point's.

Though the above restriction seems to be reasonable, it is sometimes too strict to support many kinds of advice needed in practice as we will discuss in Section 3. In fact, AspectJ and StrongAspectJ[7] relax the restriction in order

* The core part of the paper was presented at the 9th International Conference on Aspect-Oriented Software Development (AOSD.10)[15]. We revised the formalization in Section 5 along with full language definition and correctness proofs.

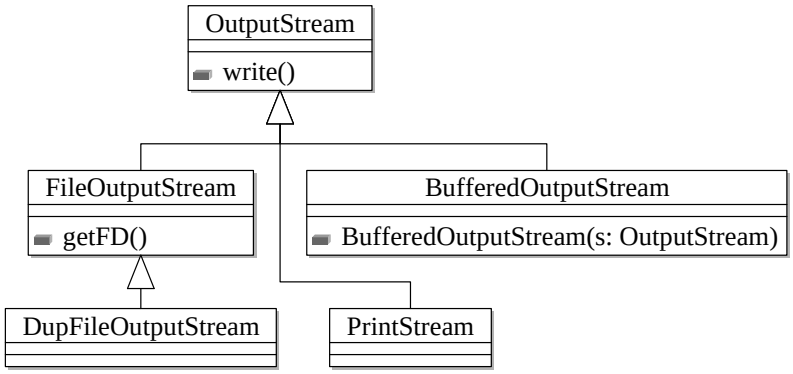

Fig. 1. UML class diagram of the classes that appear in the examples.

to support generic advice declarations that are applied to join points of different types. AspectJ's mechanism gives a special meaning to `Object` type in around advice declarations at the risk of runtime cast errors, whereas StrongAspectJ's mechanism guarantees type safety.

In this paper, we present an approach that relaxes the restrictions by giving different kind of type genericity to around advice. While existing languages type-check with respect to types of join points, our proposed mechanism checks how the values supplied from around advice are used in subsequent computation. This enables to weave pieces of around advice that have been rejected in existing languages, while guaranteeing type safety. Examples of such pieces of advice include the one that replaces an object with another object of a sibling class, and the one that inserts a wrapper to an anonymous class.

In the rest of the paper, we first introduce around advice in AspectJ in Section 2. We then, in Section 3, show that several useful around advice declarations are rejected in AspectJ due to its type-checking rules. Section 4 proposes our new mechanism called *type relaxed weaving*, that accepts such around advice declarations. The section also discusses subtle design decisions to guarantee type safety. We formalized a core part of the mechanism to show its type soundness in Section 5. We also implemented the mechanism as an AspectJ compatible compiler called *RelaxAJ*, as described in Section 6. After discussing related work in Section 7, Section 8 concludes the paper. The detailed language definitions and proofs are presented in the appendices.

2 Around Advice in AspectJ

This section introduces the around advice mechanism in AspectJ by using an example where around advice in AspectJ works well. Readers familiar with its type-checking rules can skip this section.

We first show a method to which we will apply pieces of around advice. We sometimes call it *a base program* or *a base method*. The `store` method (shown in

Listing 1.1) stores graphical data into a file in a graphical drawing application.[3] It is executed when the user selects the save menu. The hierarchy of the relevant classes is summarized in Figure 1.

```
void store(String fileName, Data d) {
  FileOutputStream s = new FileOutputStream(fileName);
  BufferedOutputStream output = new BufferedOutputStream(s);
  output.write(d.toByteArray());
  output.close();
}
```

Listing 1.1. A method that stores graphical data into a file.

Assume we want to duplicate every file output to the console for debugging purposes. Without AOP, this can be achieved by replacing "`new FileOutputStream(fileName)`" in the second line of the `store` method with "`new DupFileOutputStream(fileName)`", where `DupFileOutputStream` is a subclass of `FileOutputStream` as defined below.

```
class DupFileOutputStream extends FileOutputStream {
  void write(int b) {
    super.write(b); System.out.write(b);
  }
  ...overrides other write methods as well...
}
```

With AOP, instead of textually editing the `store` method, we can write an around advice declaration that creates `DupFileOutputStream` objects instead of `FileOutputStream`, as shown in Listing 1.2. In AspectJ, advice declarations are written in aspect declarations, which we omit in the paper for simplicity.

```
DupFileOutputStream around(String n):
    call(FileOutputStream.new(String)) && args(n) {
  return new DupFileOutputStream(n);
}
```

Listing 1.2. A piece of advice that creates `DupFileOutputStream` objects instead of `FileOutputStream` objects.

The advice declaration begins with a *return type* (`FileOutputStream`), which specifies a type of values returned by the advice, followed by the `around` keyword.

[3] The method is taken from JHotDraw (`http://www.jhotdraw.org/`) version 6.0b1, but simplified for explanatory purposes.

Between the subsequent parentheses, there are formal parameters to the advice (`String n`). The next element is a *pointcut*, which specifies when the advice will run. The pointcut in Listing 1.2 specifies constructor calls to the `FileOutputStream` class, and also binds the constructor parameter to the variable `n` when it matches. Finally, a *body* of the advice, which consists of Java statements, is written in the braces.

Execution of a base program with pieces of around advice can be explained in terms of *join points*, which represent the operations in program execution whose behavior can be affected by advice. In AspectJ, the join points are the operations about objects, including method and constructor calls, method and constructor executions, field accesses, and exception throwing and catching operations. It should be noted that the operations about local variables are not included.

When a program is to execute an operation, we say the program creates a join point corresponding to the operation. When there is any around advice declaration that has a pointcut specifying the join point, the body of the advice runs instead of the join point. For example, when the `store` method runs, at line 2, it creates a join point corresponding to a constructor call to `FileOutputStream`. Since the join point is specified by the pointcut of the advice in Listing 1.2, the program creates a `DupFileOutputStream` object instead of `FileOutputStream`. If there is more than one around advice declaration matching a join point, one of them, selected by a certain rule, will run. Calling a pseudo-method `proceed` in the body of around advice lets the next around advice run, if there is one. Otherwise, the operation corresponding to the original join point is executed.

Practical AOP languages *weave* advice declarations into a base program. AspectJ, for example, weaves advice declarations in two steps. First, it compiles class definitions into bytecode as if they are pure Java definitions. It also transforms a body of each advice declaration into a Java method in a bytecode format. Second, it examines bytecode instructions generated at the first step. For each sequence of instructions that creates join points at runtime, which is called a *join point shadow*, if there is an advice declaration that specifies join points created from the shadow, it replaces the instructions with a call instruction to a method translated from the advice body. When the advice has a *dynamic pointcut*, which specifies join points by using runtime information, AspectJ inserts a series of instructions that evaluate the runtime condition, and then branch to either a call to the method translated from the advice body, or a call to the original method.

For the example in Listing 1.1, the expression `new FileOutputStream(fileName)` is compiled into a sequence of bytecode instructions that creates and initializes an object, which is a join point shadow. Since the shadow creates constructor-call join points that are specified by the pointcut of the around advice in Listing 1.2, the compiler replaces the instructions with a call to a method that is generated from the body of the around advice, which creates a `DupFileOutputStream` object.

3 Type-Checking Rules and their Problems

3.1 The Rules

AspectJ compilers type-check around advice on the following two regards: (1) each advice body should be consistent with a return type of the advice declaration; and (2) the return type of an advice declaration should be consistent with join point shadows where the advice is woven into. Below we present more detailed rules.[4]

The first rule is similar to the rule for a return statement in a method.

Rule 1 (Advice Body) *Any return statement in the body of around advice must have a subtype*[5] *of the return type of the advice declaration.*

After checking this rule, we can trust the advice's return type. An advice body that does not match the return type like the one shown below will be rejected.

```
FileOutputStream around():
    call(FileOutputStream.new(String)) {
  return "not a stream";                // type error
}
```

The second rule ensures that around advice will be woven into "right" join point shadows.

Rule 2 (AspectJ Weaving) *The return type of an around advice declaration must be a subtype of the return type of any join point shadow where the advice is woven into.*

For example, AspectJ allows to weave the around advice in Listing 1.2 into an expression `new FileOutputStream(fileName)` because the return type of the advice (i.e., `DupFileOutputStream`) is a subtype of the return type of the join point shadow (i.e., `FileOutputStream`).

However, AspectJ reports an error when it attempts to weave the following into `new FileOutputStream(fileName)`.

```
String around():                        // weave error
    call(FileOutputStream.new(String)) {
  return "not a stream";
}
```

[4] The rules are guessed by the authors from behavior of existing compilers, which are consistent with the ones presented by De Fraine, et al.[7], except that we omit the rules for ad-hoc generic typing with `Object`.

[5] In the paper, subtype and supertype relations are reflexive: i.e., for any `T`, `T` is a subtype and supertype of `T`.

This is because, the return type of the advice (i.e., `String`) is not a subtype of the return type of the join point shadow (i.e., `FileOutputStream`).

Note that AspectJ rejects the above advice when it weaves the advice into a join point shadow, but not when it checks the advice declaration itself. In other words, AspectJ does not use the pointcut expression when it checks the advice declaration itself, even if it suggests that the advice will be woven into join points of type `FileOutputStream`.

3.2 An Example of the Problem: Replacing with a Sibling

The rules in AspectJ are too restrictive and sometimes prohibit to define useful advice. For example, assume we want to redirect the output to the console, instead of duplicating. At first, it would seem easy to define a piece of advice that returns `System.out`, which represents a stream to the console, at the creation of a `FileOutputStream` object. However, it is impossible to straightly define such advice in AspectJ.

Listing 1.3 defines four pieces of advice, which try to return `System.out` whenever a program is to create a `FileOutputStream` object.

```
PrintStream around():                    // weave error in AspectJ
    call(FileOutputStream.new(String)) {
  return System.out;                     // of type PrintStream
}

OutputStream around():                   // weave error in AspectJ
    call(FileOutputStream.new(String)) {
  return System.out;                     // of type PrintStream
}

FileOutputStream around():
    call(FileOutputStream.new(String)) {
  return System.out;                     // type error
}

Object around():                         // compiles, yet produces
    call(FileOutputStream.new(String)) {// runtime cast error in
  return System.out;                     // the store method
}
```

Listing 1.3. Pieces of advice that returns the `PrintStream` object (i.e., the console) instead of creating `FileOutputStream`.

Even though the definitions look reasonable, none of them work because of the following reasons.

– RULE 2 rejects the first and second declarations because the return types of the advice (i.e., `PrintStream` and `OutputStream`) are not a subtype of the join point shadow's return type (i.e., `FileOutputStream`, as shown in Figure 1).
– RULE 1 reject the third declaration because the type of the `return` statement (i.e., `PrintStream`) is not a subtype of the return type of the declaration (i.e., `FileOutputStream`).
– AspectJ compiles the fourth declaration without errors, but the woven program yields a runtime cast error at the second line of the `store` method (Listing 1.1) because the type of the return value is not a subtype of `FileOutputStream`.

Interestingly, if we can edit the body of the method `store`, the redirection could be easily achieved by replacing the second line of `store` (Listing 1.1)

```
FileOutputStream s = new FileOutputStream(fileName);
```

with the next one.

```
OutputStream s = System.out;// of type PrintStream
```

3.3 Another Example: Wrapping Anonymous Classes

Another example of the problem can be found when we want to wrap handler objects. Java programs frequently use instances of anonymous class[8, Section 15.9] as handler objects, i.e., objects that define call-back functions. For example, the next code fragment creates two button objects in the Swing library and a listener object, and then registers the listener object with the button objects. The listener object belongs to an anonymous class that implements the `ActionListener` interface. The parameter type of the `addActionListener` method is `ActionListener`.

```
JButton b1 = new JButton();
JButton b2 = new JButton();
ActionListener a = new ActionListener () {
  void actionPerformed(ActionEvent e) { ... }
};
b1.addActionListener(a);
b2.addActionListener(a);
```

Now, assume that we want to wrap the listener object with a `Wrapper` object whose definition is shown in the first half of Listing 1.4.

Even though we want to define a piece of advice as shown at the last half of the Listing 1.4, RULE 2 rejects it because the return type of the advice (i.e., `ActionListener`) is not a subtype of the return type of the join point, which is an anonymous class that implements `ActionListener`. Since the return type of the join point is anonymous, there is no way to declare a piece of around advice to that join point!

```
class Wrapper implements ActionListener {
  Wrapper(ActionListener wrappee) { ... }
  ...
}
ActionListener around():                    //weave error in AspectJ
    call(ActionListener+.new(..)) {
  ActionListener l = proceed();
  return new Wrapper(l);
}
```

Listing 1.4. A wrapper class and an advice declaration that wraps `ActionListener`s.

program name	Javassist	ANTLR	JHotDraw	jEdit	Xerces
program size (KLoC)	43	77	71	140	205
number of shadows	862	1,827	3,558	8,524	3,490
supertype(%)	177 (21)	315(17)	576 (16)	2,499(29)	650(19)
subtype(%)	37 (4)	70 (4)	170 (5)	974(11)	156 (4)
unrelated(%)	0 (0)	4 (0)	42 (1)	42 (0)	64 (2)
same(%)	648 (75)	1,438(79)	2,770 (78)	5,009(59)	2,620(75)

Table 1. Classification of join point shadows by the usage of the supplied values in practical applications.

Note that the advice is applied to the constructions of anonymous class objects thanks to the plus sign in the pointcut description, which specifies any subtype of `ActionListener`.

Again, wrapping can be easily done if we edited the original code fragment as follows.

```
ActionListener a = new Wrapper(
  new ActionListener () {
    void actionPerformed(ActionEvent e) { ... }
  }
);
b1.addActionListener(a);
b2.addActionListener(a);
```

3.4 Relaxation Opportunities

We carried out preliminary assessment to investigate how likely the problem mentioned above can happen in practical application programs. We counted, in practical application programs, the number of such join point shadows whose return value is used merely as its strict supertype of the type of the shadows. As we see in the examples in Sections 3.2 and 3.3, the problem only happens at such join point shadows. In other words, if there are only few such join point shadows in programs, the problem is unlikely to happen. Since the assessment does not

consider existence of useful advice in practice, it merely supports chances of existence of the problem.

We examined five medium-sized Java programs and classified relations of a static type of a join point shadow in the programs. Classification is based on the most general type among static types in the operations that use the value from the shadow. For the ease of examination, we identified the dataflow relations by writing an aspect in AspectJ that dynamically monitors program executions. Note that the results are approximation as we used a dynamic monitoring technique.

The evaluated programs and the results are summarized in Table 1. The *supertype* row shows the numbers of shadows whose return values are used only as strict supertypes of the return type in subsequent computation in the same method execution. Similarly, the *subtype, unrelated,* and *same* rows show the number of shadows classified by the types in the operations using the returned values[6]. The numbers on the supertype row, which are approximately 15–30% in the table, suggest that there are code fragments in practical applications in which the problem presented in the previous sections can happen.

3.5 Generality of the Problem

While the examples are about return values from around advice in AspectJ, the problem is not necessarily limited to those cases.

First, the problem is not limited to return values. Since around advice can also replace parameter values that are captured by `args` and `target` pointcuts, the same problem can happen at replacing those values by using the `proceed` mechanism. For example, if the `store` method in Listing 1.1 takes `FileOutputStream` rather than a file name string, the following around advice requires relaxed weaving rules[7].

```
void around(OutputStream s): args(s,*) &&
    execution(void store(FileOutputStream,Data)) {
  proceed(System.out);
}
```

In this paper, however, we only consider types about return values, and leave types of parameter values to future work.

Second, the problem is not limited to AspectJ. Statically typed AOP languages that support around advice, such as CaesarJ[17] and AspectC++[19], should also have the same problems.

Third, the problem is not limited to AOP languages. The same problem would arise with the language mechanisms that can intercept and replace values, such as method-call interception[13] and type safe update programming[6].

[6] The subtype and unrelated relations appear in programs that have downcasting and more than one interface type, respectively.

[7] Interestingly, an AspectJ compiler (ajc version 1.5.3) compiles the example. However, the generate code does not work as it contains a cast operation to `FileOutputStream`.

4 Type Relaxed Weaving

4.1 Basic Idea

We propose a weaving mechanism, called *type relaxed weaving*, to address the problem while preserving type safety. Roughly speaking, it replaces the Rule 2 with the following one.

Rule 3 (Type Relaxed Weaving) *When a piece of around advice is woven into a join point shadow, the return type of the advice must be consistent with the operations that use the return value from the join point shadow.*

We here used ambiguous terms such as "consistent with" and "operations," which shall be elaborated in the later sections.

For example, the rule allows the first advice declaration in Listing 1.3 to be woven into the `store` method (Listing 1.1). The return type of the advice (`PrintStream`) is consistent with the operations that use the return value, namely the `new BufferedOutputStream(s)` expression at line 3 in Listing 1.1.

4.2 Design Principles

We assumed the following principles in designing our weaving mechanism.

Preservation of object interfaces: The mechanism should not change the signatures of methods and fields when weaving advice declarations into a program. Changing method/field signatures would give further freedom to advice, but is problematic with unmodifiable libraries and programs that use the reflection API.

Bytecode-level weaving: The mechanism weaves advice declarations into a bytecode program, in a similar way that most AspectJ compilers do. The mechanism, therefore, can merely use type information available in a bytecode base program. At the same time, the mechanism can generate woven programs that cannot be generated from Java source code, as we shall discuss in Section 4.6.

Compatibility with AspectJ: The mechanism should accept a program that is accepted by existing AspectJ compilers and should yield executable code that has the same behavior as the one compiled by existing AspectJ compilers.

Independence of advice precedence: Given advice declarations, the mechanism judges whether it is type-safe to weave them, regardless of their precedence. This makes the mechanism simpler with more predictable behavior.

4.3 Overview

The type relaxed weaving has the same mechanism as the ones in existing AOP languages except for the part that type-checks advice to be woven. Our mechanism checks type safety of advice in the following steps.

First, as explained in Section 2, the weaver processes each method in the base program. It looks for advice declarations that are applicable to join point shadows in the method.

Second, for each join point shadow, it performs dataflow analysis to identify operations that use a return value from the shadow. The extent of the analysis is within the method: i.e., it is an intra-procedural dataflow analysis. The next subsection will discuss our choice of operations.

Finally, it checks whether the return types of around advice applicable to the shadows are consistent with the operations that use a return value. If there is a piece of around advice whose return type that is not consistent with an operation, it rejects the advice as a type error.

Below, we discuss several issues of defining concrete weaving rules.

4.4 Operations that Use Return Values

Basically, operations that use values are determined by the semantics of the Java bytecode language and the principle of the interface preservation.

Below, we summarize the operations that use a value (denoted as `v`) returned from a join point shadow. Since we use a dataflow analysis, `v` actually denotes any variable that has a dataflow relation from a join point shadow.

Method or constructor call parameter: A method call `o.m(v)` uses `v` as of the parameter type in the signature of `m`. Similarly, a constructor call `new C(v)` uses `v` as of the parameter type in the signature of the constructor.
Note that the method and constructor signatures are determined before weaving. In other words, even if a piece of around advice changes types of some values, it will not change the selection of overloaded methods and constructors.

Method call target: A method call `v.m()` uses `v` as of the target type in the signature of `m`. There are subtle issues of selecting a target type. We will discuss it in Section 4.5.

Return value from a method: A return statement `return v` uses `v` as of the return type of the method.

Field access target: A field assignment `v.f=...` or a field reference `v.f` uses `v` as of the class that declares `f`.

Assigned value to a field: A field assignment `o.f = v` uses `v` as of the `f`'s type.

Array access target: An array assignment `v[i]=...` or an array reference `v[i]` uses `v` as an array type. The type of the elements is determined by the results of a dataflow analysis on the assigned or referenced value. For AspectJ compatibility, we regard all the reference types (i.e., classes, interfaces and arrays) as the same type when they appear as an element type of an array.

Exception to throw: A throw statement `throw v` uses `v` as one of the types in the `throws` clause of the method declaration, or a type in one of `catch` clause around the `throw` statement.

Note that the above operations do not include accesses to a local variable. This gives the mechanism more opportunity to weave advice declarations with more general types. For example, in Listing 1.1, even when the return value from `new FileOutputStream(fileName)` is stored in a variable of type `FileOutputStream` at line 2, the return value is not considered as used by the assignment. Only the subsequent operation `new BufferedOutputStream(s)` at line 3 is regarded as the operation that uses the value.

Ignoring types of local variables also fits for current AspectJ compilers, which support bytecode weaving. In Java bytecode, local variables have no static types in a sense that can affect behavior of a program; types of local variables are merely available as hints for debuggers.

4.5 Target Type of a Method Call

When a value is used as a target of a method call, we should examine a type hierarchy to decide the types that the value is used as. This is because, even though there is target type information in a Java bytecode instruction, the static type of the target object can be any subtype of a supertype of the target type, as long as the supertype declares the method.

For example, assume a method call `o.write(...)` whose target type is `BufferedOutputStream`. Since `write` is declared in `OutputStream` (as shown in Figure 1), it is safe to change the target type of the call to `OutputStream`. The static type of `o` can thus be any subtype of `OutputStream`.

Therefore, when a value is used as a target of a method call `m`, we regard the value is used as a value of the most general type that declares `m`, rather than the target type in the signature of `m`. This will give our weaver a chance to accept more advice.

However, when there are interface types, we need to consider that a target object is used as a value of one of several types, rather than a single type. Consider the interfaces and class declarations followed by a code fragment shown in Listing 1.5.

```
interface Runnable { ... }
interface Task     { void show();/* displays the task's status */ }
class Simulator    { void show() { ... } }
class BkSimulator extends Simulator
                  implements Runnable,Task { ... }
// in a method:
BkSimulator o = new BkSimulator();
new Thread(o).start();
...
o.show(); //signature: void BkSimulator.show()
```

Listing 1.5. A method call with a class and interface types.

At the bottom line, we should regard that `o` is used as a value of *either* `Simulator`, `BkSimulator`, or `Task`, regardless of the target type in `show`'s signature. Since there is no subtype relation between `Simulator` and `Task`, we should at least consider these two types to maximize weaving opportunity.

Note that *covariant return types*[8, Section 8.4] in Java, which allows an overriding method in a subclass to have a more specific return type than the one in the overridden method, can complicate the safety condition. Our current approach here is to rely on the types in compiled bytecode, where signatures of two methods—an overriding method with a covariant return type and the method to be overridden—are regarded as different (i.e., not overriding).

4.6 Usage as a Value of Multiple Interface Types

Existence of interface types also requires to consider that an object is used as a value of a subtype of several unrelated types.

In Listing 1.5, the constructor `new Thread(o)` has a parameter of type `Runnable`. Therefore, we should consider that `o` is used as a value of a subtype of *both* `Runnable` and `Task`, in order to increase a weaving opportunity.

This means that, when there is a class that implements both `Runnable` and `Task`, we can replace the return value from `new BkSimulator()` with an object of such a class even if it is not a subtype of `BkSimulator`. The following declarations demonstrate the case.

```
class RunnableTask implements Runnable,Task { ... }
RunnableTask around(): call(BkSimulator.new()) {
  return new RunnableTask();
}
```

When we weave this advice into Listing 1.5, we also need to change the target type of the method invocation instruction for the last method call from `BkSimulator` to `Task`.

Note that application of the above around advice to Listing 1.5 could generate woven code that cannot be generated from the Java source language when the pointcut contains a dynamic condition, such as `if(debugging)`. In this case, the woven code for the line

```
BkSimulator o = new BkSimulator();
```

will become the one like below, if translated back into the source language.

```
if (debugging) o = new RunnableTask();
else           o = new BkSimulator();
```

Even though it is not a valid source program because we cannot give a static type for `o`, the woven code is valid at the bytecode language level. We refer readers to formalizations of the Java bytecode language[14] for a detailed discussion.

4.7 Relaxation within Base Programmer's Expectation

Even though we might be able to radically relax types as shown in the previous section, we confine return types so as that the woven program's behavior stays within expectation of the base programmers.

Consider the following interface, class and a piece of advice.

```
interface Presentation { void show(); void stop(); }
class SlideSet implements Runnable,Presentation { ... }
SlideSet around(): call(BkSimulator.new()) {
  return new SlideSet();
}
```

It is safe to apply the above advice to Listing 1.5 because a `SlideSet` object can be used as a parameter to the constructor of `Thread(Runnable)`, and can be used as a target of a method `show()`.

Even if it is safe, weaving it into Listing 1.5 could violate the base programmer's expectation because the `show()` method specified in the `Presentation` interface might accidentally share the same name with the one in the `Task` interface. Similar to the problem of name collision with multiple inheritance[12], a class that accidentally has the same name method with another class might not be semantically compatible.

We therefore allow a piece of around advice whose return value will be used as a target of a method call only if there exists a common supertype between the advice's return type and the target type of the method call that declares the method. With this rule, the advice that returns `RunnableTask` (Section 4.6) can be applied to Listing 1.5 because `RunnableTask` is subtype of `Task`, and the target type of `o.show()` in Listing 1.5 is `BkSimulator`, which is subtype of `Task` as well. However, the advice that returns `SlideSet` is not allowed because there is no common supertype between `SlideSet` and `BkSimulator` that defines `show()`.

Note that the rule is consistent with dynamic pointcuts. As explained in Section 4.6, around advice is conditionally executed when its pointcut contains a dynamic condition. If the return type of a piece of around advice satisfies the above rule, we can always find a target type of a method call even if the return value is stored into the same variable with the return value from the join point.

4.8 Multiple Advice Applications at a Join Point

When the mechanism checks usage of a return value from a join point, it should also check the operations in advice bodies that are applied to the same join point. This is because, when there is more than one piece of advice applied to a join point, one of the pieces can use a return value from another piece by executing a `proceed` operation. Since our principle is not to rely on advice precedence, the mechanism should take all pieces of advice applied to the same join point into account.

For example, assume that the following advice is applied to the `store` method (in Listing 1.1) in conjunction with the first advice declaration in Listing 1.3.

```
FileOutputStream around():
    call(FileOutputStream.new(String)) {
  FileOutputStream s = proceed();
  s.getFD().sync();//uses s as FileOutputStream
  return s;
}
```

Our weaver then has to reject the combination of the above advice and the first one in Listing 1.3. This is because, when the above advice runs prior to the first advice in Listing 1.3, the former advice receives a return value from the latter, and uses the value as a `FileOutputStream` object by calling the `getFD` method (see Figure 1).

4.9 Advice Interference

When there are advice declarations applicable to more than one join point shadow in a method, it should check consistency among all those shadows in a method at once. In other words, it is not possible to employ an approach to safety checking in existing weavers, which check safety for each join point shadow independently.

The following example demonstrates a case in which two advice declarations cannot coexist, even though each of them can be safely applied without the other.

The case consists of two interfaces `I` and `J`, two classes `C` and `D` that implement both interfaces, followed by a code fragment that assigns objects in different classes into one variable.

```
interface I { C m(); }
interface J { C m(); }
class C implements I,J { C m(){ ... } }
class D implements I,J { C m(){ ... } }
// in a method:
I x;
if (...) x = new C(); else x = new D();
x.m(); // uses x as a value of I or J
```

Now consider the following class and advice declarations that replace creations of `C` and `D` objects with those of `E` and `F` objects, respectively.

```
class E implements I   { C m(){ ... } }
class F implements   J { C m(){ ... } }
E around(): call(C.new()) { return new E(); }//CtoE
F around(): call(D.new()) { return new F(); }//DtoF
```

Application of both advice declarations is not correct because we cannot give a single target type for `x.m()`. Applications of either one of above two advice declarations are, however, safe.

5 Formal Correctness of Type Relaxed Weaving

In this section, we formalize a core of the type relaxed weaving to confirm its type safety. We first set up a simple object-oriented language called Featherweight Java for Relaxation (FJR), which is an extension of Featherweight Java (FJ)[9]. The language models programs after advice is woven, by adding advice application and `proceed` to FJ; its type system, which we believe corresponds to bytecode verification in the Java Virtual Machine, is sound with respect to operational semantics. We then present a procedure to type-check a woven program; it takes an expression (with its type environment) and returns its type (if it is well typed) and a new expression where type annotation has been changed. We prove that the type-checking procedure is correct with respect to the typing rules and that the new expression is "similar" to the original expression before weaving in a certain sense.

We show only a main part of definitions and the statements of theorems in this section and refer readers to Appendices for the full definitions and proofs.

5.1 Featherweight Java for Relaxation

Syntax Featherweight Java for Relaxation has, in addition to most of the features in FJ, interface types (without interface extension), let expressions, nondeterministic choice, woven advice, and `proceed`. However, for simplicity reasons, we remove typecasts and fields from objects, which can be easily restored without difficulty. Unlike FJ, where every expression is assigned a class name as its type, FJR has a restricted version of the union type system[10], which allows more liberal use of values as discussed in Section 4.6.

The syntax rules of FJR are given as follows.

$$
\begin{array}{rll}
\texttt{CL} ::= & \texttt{class C extends C implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} & \textit{class} \\
\texttt{M} ::= & \texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} & \textit{method} \\
\texttt{IF} ::= & \texttt{interface I \{ } \overline{\mathcal{S}} \texttt{ \}} & \textit{interface} \\
\mathcal{S} ::= & \texttt{T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{);} & \textit{signature} \\
\texttt{a}, \texttt{b} ::= & \texttt{T(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} & \textit{advice} \\
\texttt{e} ::= & \texttt{x} \mid \texttt{e}_{\langle \texttt{T} \rangle}\texttt{.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C()} \mid \texttt{let x = e in e} & \textit{expression} \\
& \mid \texttt{(?e:e)} \mid \texttt{[}\overline{\texttt{a}}\texttt{](}\overline{\texttt{e}}\texttt{)} \mid \texttt{proceed(}\overline{\texttt{e}}\texttt{)} & \\
\texttt{S}, \texttt{T} ::= & \texttt{C} \mid \texttt{I} & \textit{simple type} \\
\texttt{U}, \texttt{V} ::= & \texttt{T} \mid \texttt{U} \cup \texttt{U} & \textit{type}
\end{array}
$$

Following the convention of FJ, we use an overline to denote a sequence and write, for example, $\overline{\texttt{x}}$ as shorthand for $\texttt{x}_1, \ldots, \texttt{x}_n$. The metavariables `C` and `D` range over class names; `I` and `J` range 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 $\overline{\texttt{M}}$; `IF` is an interface declaration, consisting of its name and method headers $\overline{\mathcal{S}}$. In addition to class declarations, FJR has interface declarations, which are needed to discuss the issues of target types (Sections 4.5 and 4.6) and advice interference (Section 4.9).

The syntax of expressions is extended from that of FJ, which already includes variables, method invocation, and object instantiation (and also field access and typecast, which are omitted here). Here, a method invocation expression $\texttt{e}_{\langle \texttt{T} \rangle}\texttt{.m(}\overline{\texttt{e}}\texttt{)}$ records the static type `T` of the receiver explicitly as in the bytecode instructions `invokevirtual` and `invokeinterface`. This information will be used during type-checking to ensure that types in the woven program are not very different from the original. We introduce `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 $\texttt{let x = e}_1\ \texttt{in}\ \texttt{e}_2$ is bound in $\texttt{e}_2$. We also introduce non-deterministic choice `(?e:e)` to handle the cases when a variable contains values of different types. Although a non-deterministic choice is not useful in practical programming, it is sufficient to express such a case.

The last two forms are related to advice execution. The first form $\texttt{[}\overline{\texttt{a}}\texttt{](}\overline{\texttt{e}}\texttt{)}$ represents an application of a sequence $\overline{\texttt{a}}$ of pieces of advice to arguments $\overline{\texttt{e}}$. In FJR, a piece of advice `a` is represented by an anonymous function, which can call the next advice by `proceed`—the last form of expressions. Actually, the last advice in $\overline{\texttt{a}}$ should be considered the original code of the join point into which advice is woven. For example, the result of weaving

```
PrintStream around():
    call(FileOutputStream.new()) {
  return System.out;
}
```

into an expression `new FileOutputStream()` would be expressed by

```
[ PrintStream(){ return System.out; },
  FileOutputStream(){ return new FileOutputStream(); }
]()
```

which will reduce to `System.out` without calling the original code.

Remark 1. Although this model of advice application may look too simple, we believe that it is sufficient to show the essence of the type relaxed weaving. We do not model a weaving algorithm (in particular, how advice is selected) here, partly because it is basically the same as AspectJ's. Actually, for type safety, it does not matter which advice is applied—type-checking will be performed for the woven code anyway. We should note, however, that this model has one significant difference from the implementation explained in the next section. This model does not express the fact that an advice body is shared among joint point shadows to which the advice is applied. So, a single advice body would have to be duplicated. As a result, one advice body might be type-checked several times under different assumptions, depending on joint point shadows to which this advice is woven into. It allows polymorphic use of a single piece of advice. We have chosen simplicity over faithfulness here.

We define the set of free variables in an expression by a standard manner. We will denote a capture-avoiding substitution of expressions $\overline{\texttt{e}}$ for variables $\overline{\texttt{x}}$

by $[\overline{\mathtt{e}}/\overline{\mathtt{x}}]$. We use the notation $[[\overline{\mathtt{a}}]/\mathtt{proceed}]\mathtt{e}$ to replace $\mathtt{proceed}(\overline{\mathtt{e}})$ in $\mathtt{e}$ with $[\overline{\mathtt{a}}](\overline{\mathtt{e}})$. More precisely, $[[\overline{\mathtt{a}}]/\mathtt{proceed}]\mathtt{e}$ is defined by:

$$
\begin{array}{ll}
[[\overline{\mathtt{a}}]/\mathtt{proceed}]\mathtt{x} & = \mathtt{x} \\
[[\overline{\mathtt{a}}]/\mathtt{proceed}](\mathtt{e}_0\langle\mathtt{T}\rangle.\mathtt{m}(\overline{\mathtt{e}})) & = ([[\overline{\mathtt{a}}]/\mathtt{proceed}]\mathtt{e}_0)\langle\mathtt{T}\rangle.\mathtt{m}([[\overline{\mathtt{a}}]/\mathtt{proceed}]\overline{\mathtt{e}}) \\
[[\overline{\mathtt{a}}]/\mathtt{proceed}](\mathtt{new\ C()}) & = \mathtt{new\ C()} \\
[[\overline{\mathtt{a}}]/\mathtt{proceed}] & = \mathtt{let\ x\ =}\ ([[\overline{\mathtt{a}}]/\mathtt{proceed}]\mathtt{e}_1) \\
\quad(\mathtt{let\ x\ =\ e_1\ in\ e_2}) & \quad \mathtt{in}\ ([[\overline{\mathtt{a}}]/\mathtt{proceed}]\mathtt{e}_2) \\
[[\overline{\mathtt{a}}]/\mathtt{proceed}](\mathtt{?e_1{:}e_2}) & = (\mathtt{?}([[\overline{\mathtt{a}}]/\mathtt{proceed}]\mathtt{e}_1)\mathtt{:}([[\overline{\mathtt{a}}]/\mathtt{proceed}]\mathtt{e}_2)) \\
[[\overline{\mathtt{a}}]/\mathtt{proceed}]([\overline{\mathtt{b}}](\overline{\mathtt{e}})) & = [\overline{\mathtt{b}}]([[\overline{\mathtt{a}}]/\mathtt{proceed}]\overline{\mathtt{e}}) \\
[[\overline{\mathtt{a}}]/\mathtt{proceed}](\mathtt{proceed}(\overline{\mathtt{e}})) & = [\overline{\mathtt{a}}]([[\overline{\mathtt{a}}]/\mathtt{proceed}]\overline{\mathtt{e}})
\end{array}
$$

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. Union types are used for return types of proceed as well as local variables.

An FJR program is a pair of a class table CT, which is a mapping from class/interface names to class and interface declarations, and an expression, which stands for the body of the main method. We denote the domain of a mapping by $dom(\cdot)$. We always assume a fixed class table, which is assumed to satisfy the following sanity conditions:

1. $CT(\mathtt{C}) = \mathtt{class\ C}\ \cdots$ for every $\mathtt{C} \in dom(CT)$;
2. $CT(\mathtt{I}) = \mathtt{interface\ I}\ \cdots$ for every $\mathtt{I} \in dom(CT)$;
3. $\mathtt{Object} \notin dom(CT)$;
4. for every simple type T (except Object) appearing anywhere in CT, we have $\mathtt{T} \in dom(CT)$; and
5. there are no cycles formed by extends clauses.

Lookup Functions As in FJ, we use the functions, whose definitions are shown only in Appendix A, to look up method signatures and bodies in the class table. $mtype(\mathtt{m}, \mathtt{T})$ returns a pair (written $\overline{\mathtt{S}} \to \mathtt{S}_0$) of the sequence of the argument types $\overline{\mathtt{S}}$ and the return type $\mathtt{S}_0$ of method m in simple type T (or its supertypes). We also use the function $mtype^C(\mathtt{m}, \mathtt{C})$, which also returns the signature of m in C; unlike *mtype*, it looks up only superclasses and do not consider interfaces that the given class implements. $mbody(\mathtt{m}, \mathtt{C})$ returns a pair (written $\overline{\mathtt{x}}.\mathtt{e}$) of the formal parameters $\overline{\mathtt{x}}$ and the body e of method m in class C. Since a method is declared only in classes, *mbody* takes only class names as its second argument. We assume Object has no methods and so none of $mtype(\mathtt{m}, \mathtt{Object})$, $mtype^C(\mathtt{m}, \mathtt{Object})$ and $mbody(\mathtt{m}, \mathtt{Object})$ is defined. We also use $rettype(\mathtt{a})$ to retrieve the return type of the advice a.

Type System The subtyping relation is written $\mathtt{U}$ <: $\mathtt{V}$ and defined by rules in Figure 2. It is reflexive, transitive, and includes extends and implements relations, as in Java. The rules S-UnionR1, S-UnionR2 and S-UnionL for

union types mean that the union type $\mathtt{U}_1 \cup \mathtt{U}_2$ is the least upper bound of $\mathtt{U}_1$ and $\mathtt{U}_2$. It is easy to show that $\cup$ is associative and commutative in the sense that $\mathtt{U}_1 \cup \mathtt{U}_2$ and $\mathtt{U}_2 \cup \mathtt{U}_1$ are subtypes of each other for any $\mathtt{U}_1$ and $\mathtt{U}_2$ and so are $(\mathtt{U}_1 \cup \mathtt{U}_2) \cup \mathtt{U}_3$ and $\mathtt{U}_1 \cup (\mathtt{U}_2 \cup \mathtt{U}_3)$.

$$\mathtt{U} \mathrel{<:} \mathtt{U} \qquad \text{(S-Refl)}$$

$$\frac{\mathtt{U}_1 \mathrel{<:} \mathtt{U}_2 \qquad \mathtt{U}_2 \mathrel{<:} \mathtt{U}_3}{\mathtt{U}_1 \mathrel{<:} \mathtt{U}_3} \qquad \text{(S-Trans)}$$

$$\mathtt{U} \mathrel{<:} \mathtt{Object} \qquad \text{(S-Object)}$$

$$\frac{\mathtt{class\ C\ extends\ D\ implements}\ \overline{\mathtt{I}}\ \{\cdots\}}{\mathtt{C} \mathrel{<:} \mathtt{D}} \qquad \text{(S-Extends)}$$

$$\frac{\mathtt{class\ C\ extends\ D\ implements}\ \overline{\mathtt{I}}\ \{\cdots\}}{\mathtt{C} \mathrel{<:} \mathtt{I}_i} \qquad \text{(S-Implements)}$$

$$\mathtt{U}_1 \mathrel{<:} \mathtt{U}_1 \cup \mathtt{U}_2 \qquad \text{(S-UnionR1)}$$

$$\mathtt{U}_2 \mathrel{<:} \mathtt{U}_1 \cup \mathtt{U}_2 \qquad \text{(S-UnionR2)}$$

$$\frac{\mathtt{U}_1 \mathrel{<:} \mathtt{U}_3 \qquad \mathtt{U}_2 \mathrel{<:} \mathtt{U}_3}{\mathtt{U}_1 \cup \mathtt{U}_2 \mathrel{<:} \mathtt{U}_3} \qquad \text{(S-UnionL)}$$

Fig. 2. FJR: Subtyping rules.

The type judgment for expressions is of the form $\Gamma; \mathtt{P} \vdash \mathtt{e} : \mathtt{U}$, read "expression `e` is given type `U` under type environment Γ and the assumption that `proceed` has type `P`." and that for advice is of the form $\mathtt{P} \vdash \mathtt{a\ OK}$, read "advice `a` is well typed provided that `proceed` has type `P`." A type environment Γ, also written $\overline{\mathtt{x}}{:}\overline{\mathtt{U}}$, is a finite mapping from variables $\overline{\mathtt{x}}$ to types $\overline{\mathtt{U}}$. A type `P` of `proceed` is either $\overline{\mathtt{T}} \to \mathtt{U}$, which means that `proceed` takes arguments of type $\overline{\mathtt{T}}$ and returns `U`, or $\bullet$, which means that `proceed` cannot be called (because the expression is in a method definition).

We show the typing rules for expressions and advice in Figure 3. Most rules are straightforward. As we have already mentioned, $\mathtt{T}_0$ in T-Invk represents the receiver's static type; a simple type has to be chosen here. So, if two classes `C` and `D` extend `Object` and implement no interfaces happen to have a method `m` of the same signature, `m` cannot be invoked on the receiver of type $\mathtt{C} \cup \mathtt{D}$. T-Let allows local variables to have union types, whereas the method typing rule, given below, allows method parameters to have only simple types. In T-Choice, the choice

$$\Gamma;\mathtt{P} \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-Var)}$$

$$\frac{\Gamma;\mathtt{P} \vdash \mathtt{e}_0 : \mathtt{U}_0 \qquad \mathtt{U}_0 \mathtt{<:}\ \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{T}} \to \mathtt{T} \qquad \Gamma;\mathtt{P} \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \overline{\mathtt{U}} \mathtt{<:}\ \overline{\mathtt{T}}}{\Gamma;\mathtt{P} \vdash \mathtt{e}_{0\langle \mathtt{T}_0 \rangle}.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{T}} \qquad \text{(T-Invk)}$$

$$\Gamma;\mathtt{P} \vdash \mathtt{new\ C()} : \mathtt{C} \qquad \text{(T-New)}$$

$$\frac{\Gamma;\mathtt{P} \vdash \mathtt{e}_1 : \mathtt{U}_1 \qquad \Gamma, \mathtt{x}{:}\mathtt{U}_1;\mathtt{P} \vdash \mathtt{e}_2 : \mathtt{U}_2}{\Gamma;\mathtt{P} \vdash \mathtt{let\ x\ =\ e}_1\ \mathtt{in\ e}_2 : \mathtt{U}_2} \qquad \text{(T-Let)}$$

$$\frac{\Gamma;\mathtt{P} \vdash \mathtt{e}_1 : \mathtt{U}_1 \qquad \Gamma;\mathtt{P} \vdash \mathtt{e}_2 : \mathtt{U}_2}{\Gamma;\mathtt{P} \vdash \mathtt{(?e}_1\mathtt{:e}_2\mathtt{)} : \mathtt{U}_1 \cup \mathtt{U}_2} \qquad \text{(T-Choice)}$$

$$\frac{\Gamma;\overline{\mathtt{T}} \to \mathtt{U} \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \qquad \overline{\mathtt{V}} \mathtt{<:}\ \overline{\mathtt{T}}}{\Gamma;\overline{\mathtt{T}} \to \mathtt{U} \vdash \mathtt{proceed}(\overline{\mathtt{e}}) : \mathtt{U}} \qquad \text{(T-Proceed)}$$

$$\frac{\mathtt{U}_0 = \bigcup_{\mathtt{a} \in \overline{\mathtt{a}}, \mathtt{b}} \mathit{rettype}(\mathtt{a}) \qquad \overline{\mathtt{S}} \to \mathtt{U}_0 \vdash \overline{\mathtt{a}}\ \mathtt{OK} \qquad \bullet \vdash \mathtt{b\ OK} \qquad \Gamma;\mathtt{P} \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \qquad \overline{\mathtt{V}} \mathtt{<:}\ \overline{\mathtt{S}}}{\Gamma;\mathtt{P} \vdash \mathtt{[}\overline{\mathtt{a}}\mathtt{,b]}\,\mathtt{(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{U}_0} \qquad \text{(T-Woven)}$$

$$\frac{\mathtt{P} = \bullet \text{ or } \overline{\mathtt{S}} \to \mathtt{V} \qquad \overline{\mathtt{x}}{:}\overline{\mathtt{S}};\mathtt{P} \vdash \mathtt{e} : \mathtt{U} \qquad \mathtt{U} \mathtt{<:}\ \mathtt{T}}{\mathtt{P} \vdash \mathtt{T(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}\ OK}} \qquad \text{(T-Advice)}$$

Fig. 3. FJR: Typing rules for expressions and advice.

expression is given the union of the types of the two subexpressions since its value is that of either $\mathtt{e}_1$ or $\mathtt{e}_2$. The rule T-Proceed is as trivial as T-Var. The rule T-Woven requires that all pieces of advice be well typed and the arguments have subtypes of their parameter types $\overline{\mathtt{S}}$. Since the last advice `b` represents the original code, as already discussed, it must not refer to `proceed` (hence $\bullet$ is used). The return type of `proceed` and the type of the whole expression is the union of the return types of all pieces of advice and the original join point shadow. The typing rule T-Advice is mostly straightforward: the body must be typed under the assumption that parameters $\overline{\mathtt{x}}$ have declared types $\overline{\mathtt{S}}$, which must be the same as the argument type of `proceed` (if given), and its type `U` must be a subtype of the declared return type `T`. Note that the return type `V` of `proceed` (if given) is not related to `T` in this rule, but the combination with T-Woven will ensure `V` to be a supertype of `T`.

The type judgment for methods is of the form `M OK IN C`, read "method `M` is well typed in class `C`." The typing rule is given as follows:

$$\frac{\overline{\mathtt{x}}{:}\overline{\mathtt{T}},\mathtt{this}{:}\mathtt{C} \vdash \mathtt{e} : \mathtt{U} \quad \mathtt{U} <: \mathtt{T}_0 \quad \mathtt{class\ C\ extends\ D\ implements}\ \overline{\mathtt{I}}\ \{\cdots\} \quad \mathit{override}(\mathtt{m},\mathtt{D},\overline{\mathtt{T}}{\to}\mathtt{T}_0)}{\mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return\ e;}\ \}\ \mathtt{OK\ IN\ C}} \quad \text{(T-Method)}$$

The method body `e` has to be well typed under the type declarations of the parameters $\overline{\mathtt{x}}$ and the assumption that `this` has type `C`. As mentioned above, the parameter types and return type have to be simple types. The predicate *override*, whose definition is shown in Appendix, checks whether `M` correctly overrides the method of the same name in the superclass `D` (if any).

Finally, the type judgment for classes is written `CL OK`, meaning "class `CL` is well typed," and the typing rule is given as follows:

$$\frac{\overline{\mathtt{M}}\ \mathtt{OK\ IN\ C} \quad \forall \mathtt{m},\mathtt{I} \in \overline{\mathtt{I}}.(\mathit{mtype}(\mathtt{m},\mathtt{I}) = \overline{\mathtt{T}}{\to}\mathtt{T}_0) \Longrightarrow (\mathit{mtype}^C(\mathtt{m},\mathtt{C}) = \overline{\mathtt{T}}{\to}\mathtt{T}_0)}{\mathtt{class\ C\ extends\ D\ implements}\ \overline{\mathtt{I}}\ \{\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \quad \text{(T-Class)}$$

It means that all methods have to be well typed and all methods declared in $\overline{\mathtt{I}}$ have to be implemented (or inherited) in `C` with the correct signature. A program is well typed when all classes in the class table are well typed and the main expression is well typed under the empty type environment.

Example 1. We show a few examples of typing using interfaces and classes in Listing 1.5 (and assuming the existence of `void`). More precisely, suppose the class table includes the following interfaces and classes

```
interface Rnbl {}          // short for Runnable
interface Tsk  { void show(); } // for Tsk
class Thrd extends Object {     // for Thread
  Rnbl o;
  void start() { .. }
}
class Sim  extends Object { void show(){ .. } }
class BSim extends Sim    implements Rnbl, Tsk { .. }
class RT   extends Object implements Rnbl, Tsk { .. }
```

and let

$$\mathtt{e} \stackrel{\text{def}}{=} \mathtt{let\ o\ =\ new\ BSim()\ in\ o}_{\langle \mathtt{BSim} \rangle}\mathtt{.show()}.$$

Then, `e` is typed under the empty type environment:

$$\frac{\dfrac{}{\bullet;\bullet \vdash \mathtt{new\ BSim()} : \mathtt{BSim}}\ \text{T-New} \qquad \mathcal{D}}{\bullet;\bullet \vdash \mathtt{e} : \mathtt{void}}\ \text{T-Let}$$

where

$$\mathcal{D} \stackrel{\text{def}}{=} \frac{\dfrac{}{\mathtt{o} : \mathtt{BSim};\bullet \vdash \mathtt{o} : \mathtt{BSim}}\ \text{T-Var} \qquad \begin{array}{c}\mathtt{BSim} <: \mathtt{BSim} \\ \mathit{mtype}(\mathtt{show},\mathtt{BSim}) = \bullet{\to}\mathtt{void}\end{array}}{\mathtt{o} : \mathtt{BSim};\bullet \vdash \mathtt{o}_{\langle \mathtt{BSim} \rangle}\mathtt{.show()} : \mathtt{void}}\ \text{T-Invk}$$

Now, we consider

$$\mathtt{e}' \stackrel{\text{def}}{=} \mathtt{let\ o\ =\ [a,b]()\ in\ o}_{\langle \mathtt{Bsim} \rangle}\mathtt{.show()}$$

obtained by replacing `new BSim()` in `e` with advice application `[a,b]()` where

$$\mathtt{a} \stackrel{\text{def}}{=} \mathtt{RT()\{\ return\ new\ RT();\ \}}$$
$$\mathtt{b} \stackrel{\text{def}}{=} \mathtt{BSim()\{\ return\ new\ BSim();\ \}}.$$

Now, `[a,b]()` is typed as follows:

$$\dfrac{\dfrac{\dfrac{}{\bullet;\bullet{\rightarrow}\mathtt{RT}\cup\mathtt{BSim}\vdash \mathtt{new\ RT()} : \mathtt{RT}}\ \text{T-Advice}}{\bullet{\rightarrow}\mathtt{RT}\cup\mathtt{Bsim}\vdash \mathtt{a\ OK}} \quad \dfrac{\dfrac{}{\bullet;\bullet\vdash \mathtt{new\ BSim()} : \mathtt{BSim}}\ \text{T-Advice}}{\bullet\vdash \mathtt{b\ OK}}}{\bullet;\bullet\vdash \mathtt{[a,b]()} : \mathtt{RT}\cup\mathtt{BSim}}\ \text{T-Woven}$$

Note that the return type if the union of `RT` and `BSim`. (This advice application, however, never returns a `BSim` object.) Unfortunately, $\mathtt{e}'$ is not quite well typed—the type judgment $\mathtt{o} : \mathtt{RT}\cup\mathtt{BSim};\bullet\vdash \mathtt{o.}_{\langle \mathtt{BSim} \rangle}\mathtt{.show()} : \mathtt{void}$ cannot be derived because the type annotation on the method invocation does not match the type of `o`. We need to replace the type annotation with `Tsk` to make it type-check. In fact,

$$\mathtt{e}'' \stackrel{\text{def}}{=} \mathtt{let\ o\ =\ [a,b]()\ in\ o}_{\langle \mathtt{Tsk} \rangle}\mathtt{.show()}$$

is well typed.

$$\dfrac{\bullet;\bullet\vdash \mathtt{[a,b]()} : \mathtt{RT}\cup\mathtt{BSim} \qquad \mathcal{D}'}{\bullet;\bullet\vdash \mathtt{e}'' : \mathtt{void}}\ \text{T-Let}$$

where

$$\mathcal{D}' \stackrel{\text{def}}{=} \dfrac{\dfrac{}{\mathtt{o} : \mathtt{RT}\cup\mathtt{BSim};\bullet\vdash \mathtt{o} : \mathtt{RT}\cup\mathtt{BSim}}\ \text{T-Var} \qquad \begin{array}{c}\mathtt{RT}\cup\mathtt{BSim} <: \mathtt{Tsk}\\ \mathit{mtype}(\mathtt{show},\mathtt{Tsk}) = \bullet{\rightarrow}\mathtt{void}\end{array}}{\mathtt{o} : \mathtt{RT}\cup\mathtt{BSim};\bullet\vdash \mathtt{o}_{\langle \mathtt{Tsk} \rangle}\mathtt{.show()} : \mathtt{void}}\ \text{T-Invk}$$

Notice the use of subtyping $\mathtt{RT}\cup\mathtt{BSim} <: \mathtt{Tsk}$. By a similar argument, the expression

```
let o = [a,b]() in
let d = new Thrd(o).start() in
o⟨Tsk⟩.show()
```

has type `void`. Note that subtyping $\mathtt{RT}\cup\mathtt{BSim} <: \mathtt{Thrd}$ is used when `o` is passed to the constructor of `Thrd`. So, `o` is used as both `Tsk` and `Thrd`.

Operational Semantics The reduction relation is of the form $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression `e` reduces to expression $\mathtt{e}'$ in one step." We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. The main reduction rules are shown in Figure 4 and readers are referred to Appendix for the other rules for congruence, that is, rules that allow a subexpression to reduce. The first rule for method invocations is essentially the same as that in FJ. The rules for `let` and choice are straightforward. In the last rule R-Advice for advice call, the first advice $\mathtt{a}_0$ is called by replacing parameters with actual arguments and `proceed` with the remaining advice.

$$\frac{mbody(\mathtt{m},\mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0}{\mathtt{new\ C()}_{\langle \mathtt{T} \rangle}.\mathtt{m}(\overline{\mathtt{e}}) \longrightarrow [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ C()/this}]\mathtt{e}_0} \quad \text{(R-Invk)}$$

$$\mathtt{let\ x\ =\ e_1\ in\ e_2} \longrightarrow [\mathtt{e}_1/\mathtt{x}]\mathtt{e}_2 \quad \text{(R-Let)}$$

$$\mathtt{(?e_1 : e_2)} \longrightarrow \mathtt{e}_i \quad \text{(R-Choice)}$$

$$\frac{\mathtt{a}_0 = \mathtt{T(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e_0;\ \}}}{\mathtt{[a_0, }\overline{\mathtt{a}}\mathtt{]\,(}\overline{\mathtt{e}}\mathtt{)} \longrightarrow [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{[}\overline{\mathtt{a}}\mathtt{]/proceed}]\mathtt{e}_0} \quad \text{(R-Advice)}$$

Fig. 4. FJR: Reduction rules.

Properties FJR enjoys the standard type soundness properties consisting of subject reduction and progress[22]. See Appendix C for proofs.

Theorem 1 (Subject Reduction). *If* $\Gamma; \mathtt{P} \vdash \mathtt{e} : \mathtt{U}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$*, then there exists some type* $\mathtt{U}'$ *such that* $\Gamma; \mathtt{P} \vdash \mathtt{e}' : \mathtt{U}'$ *and* $\mathtt{U}' <: \mathtt{U}$.

Theorem 2 (Progress). *If* $\bullet; \bullet \vdash \mathtt{e} : \mathtt{U}$*, then* e *is either* `new C()` *for some* C *or there exists some expression* $\mathtt{e}'$ *such that* $\mathtt{e} \longrightarrow \mathtt{e}'$.

5.2 Type-Checking Procedure

Having set up the language, we present the type-checking procedure *TC* for FJR programs and show its correctness. The type-checking procedure takes as an input an expression after advice has been woven (with types of its free variables), even though an implementation would defer actual weaving (that is, bytecode editing) until type-checking has been done. In general, type annotations on method invocations have to be changed in order for the program to remain well typed after type relaxed weaving. To model this fact, *TC* returns another expression with its type. The expression that *TC* returns is different from the input only in its type annotations on method invocations. Moreover, the new type annotations will be supertypes of the original. We will prove that, if *TC* succeeds, then the output is well typed with respect to the typing rules and similar to the input in this sense.

Figures 5 and 6 show the type-checking procedure. It is basically obtained by reading the typing rules, which are syntax-directed, in a bottom-up manner. The only interesting case is when the input expression is a method invocation, in which case type annotation may have to be changed. Given the (new) type $\mathtt{U}_0$ of the receiver and annotation T, the new type annotation S is chosen from common supertypes, which have the method being invoked, of $\mathtt{U}_0$ and T. The returned expression has S as its type annotation. Note that we leave unspecified how S is found—finding S is decidable since, given a class table, the set of valid simple

types is finite—and which type should be chosen if more than one simple type satisfies the condition—in fact, any type is appropriate. (So, strictly speaking, TC is a relation that specifies behavior of sensible type-checking procedures.)

For example, $TC(\bullet, \bullet, \mathtt{e}') = (\mathtt{e}'', \mathtt{void})$ (where

$$\mathtt{e}' = \mathtt{let\ o\ =\ [a;\ b]()\ in\ o}_{\langle \mathtt{BSim} \rangle}\mathtt{.show()}$$
$$\mathtt{e}'' = \mathtt{let\ o\ =\ [a;\ b]()\ in\ o}_{\langle \mathtt{Tsk} \rangle}\mathtt{.show()}$$

from Example 1).

$\boxed{TC(\Gamma, \mathtt{P}, \mathtt{e}) = (\mathtt{e}', \mathtt{U})}$

$TC(\Gamma, \mathtt{P}, \mathtt{x}) = (\mathtt{x}, \Gamma(\mathtt{x}))$

$TC(\Gamma, \mathtt{P}, \mathtt{let\ x\ =\ e}_1\ \mathtt{in\ e}_2) =$
  let $(\mathtt{e}_1{}', \mathtt{U}_1) = TC(\Gamma, \mathtt{P}, \mathtt{e}_1)$ **in**
  let $(\mathtt{e}_2{}', \mathtt{U}_2) = TC((\Gamma, \mathtt{x}{:}\mathtt{U}_1), \mathtt{P}, \mathtt{e}_2)$ **in**
  $(\mathtt{let\ x\ =\ e}_1{}'\ \mathtt{in\ e}_2{}', \mathtt{U}_2)$

$TC(\Gamma, \mathtt{P}, \mathtt{e}_{0\langle \mathtt{T} \rangle}\mathtt{.m(e}_1\mathtt{,}\ \cdots\ \mathtt{,e}_n\mathtt{)}) =$
  let $(\mathtt{e}_0{}', \mathtt{U}_0) = TC(\Gamma, \mathtt{P}, \mathtt{e}_0)$ **in**
  let $(\mathtt{e}_1{}', \mathtt{U}_1) = TC(\Gamma, \mathtt{P}, \mathtt{e}_1)$ **in**
    $\vdots$
  let $(\mathtt{e}_n{}', \mathtt{U}_n) = TC(\Gamma, \mathtt{P}, \mathtt{e}_n)$ **in**
  let $\overline{\mathtt{T}} \to \mathtt{T}_0 = \mathit{mtype}(\mathtt{m}, \mathtt{T})$ **in**
  let S be a simple type such that $\mathtt{U}_0 \cup \mathtt{T}$ <: S and $\mathit{mtype}(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{T}} \to \mathtt{T}_0$ **in**
  if $\overline{\mathtt{U}}$ <: $\overline{\mathtt{T}}$ **then** $(\mathtt{e}_0{}'_{\langle \mathtt{S} \rangle}\mathtt{.m(e}_1{}'\mathtt{,}\ \cdots\ \mathtt{,e}_n{}'\mathtt{)}, \mathtt{T}_0)$ **else** *error*

$TC(\Gamma, \mathtt{P}, \mathtt{new\ C()}) = (\mathtt{new\ C()}, \mathtt{C})$

$TC(\Gamma, \mathtt{P}, \mathtt{(?e}_1\mathtt{:e}_2\mathtt{)}) =$
  let $(\mathtt{e}_1{}', \mathtt{U}_1) = TC(\Gamma, \mathtt{P}, \mathtt{e}_1)$ **in**
  let $(\mathtt{e}_2{}', \mathtt{U}_2) = TC(\Gamma, \mathtt{P}, \mathtt{e}_2)$ **in**
  $(\mathtt{(?e}_1{}'\mathtt{:e}_2{}'\mathtt{)}, \mathtt{U}_1 \cup \mathtt{U}_2)$

Fig. 5. Type-checking procedure (1)

To state correctness of TC, we introduce two relations $\mathtt{e} \hookrightarrow \mathtt{e}'$ and $\mathtt{e} \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e}'$ between expressions. The former models the fact that $\mathtt{e}'$ is a result of weaving advice into $\mathtt{e}$ (without type relaxation). The main rule is

$$\frac{\overline{\mathtt{b}} \hookrightarrow \overline{\mathtt{b}}' \qquad \overline{\mathtt{e}} \hookrightarrow \overline{\mathtt{e}}'}{\mathtt{[b]}(\overline{\mathtt{e}}) \hookrightarrow \mathtt{[}\overline{\mathtt{a}}\mathtt{,b']}(\overline{\mathtt{e}}')}$$

which allow advice weaving. The other rules, which are shown in Appendix B, are for congruence. For example, the rule for method invocations is shown below.

$TC(\Gamma, \overline{\mathtt{S}}{\to}\mathtt{U}, \mathtt{proceed(e_1\,,\,\cdots\,,e_n)}) =$
  let $(\mathtt{e_1}', \mathtt{U_1}) = TC(\Gamma, \overline{\mathtt{S}}{\to}\mathtt{U}, \mathtt{e_1})$ **in**
    $\vdots$
  let $(\mathtt{e_n}', \mathtt{U_n}) = TC(\Gamma, \overline{\mathtt{S}}{\to}\mathtt{U}, \mathtt{e_n})$ **in**
  if $\overline{\mathtt{U}}$ <: $\overline{\mathtt{S}}$ **then** $(\mathtt{proceed(e_1'\,,\,\cdots\,,e_n')}, \mathtt{U})$ **else** *error*

$TC(\Gamma, \mathtt{P}, \mathtt{[a_1\,,\,\cdots\,,a_n]\,(d_1\,,\,\cdots\,,d_m)}) =$
  let $\mathtt{T_1(\overline{S}\ \overline{x})\{\ return\ e_1;\ \}} = \mathtt{a_1}$ **in**
    $\vdots$
  let $\mathtt{T_n(\overline{S}\ \overline{x})\{\ return\ e_n;\ \}} = \mathtt{a_n}$ **in**
  let $\mathtt{U_0} = \mathtt{T_1} \cup \cdots \cup \mathtt{T_n}$ **in**
  let $(\mathtt{e_1}', \mathtt{U_1}) = TC(\overline{\mathtt{x}}{:}\overline{\mathtt{S}}, \overline{\mathtt{S}}{\to}\mathtt{U_0}, \mathtt{e_1})$ **in**
    $\vdots$
  let $(\mathtt{e_{n-1}}', \mathtt{U_{n-1}}) = TC(\overline{\mathtt{x}}{:}\overline{\mathtt{S}}, \overline{\mathtt{S}}{\to}\mathtt{U_0}, \mathtt{e_{n-1}})$ **in**
  let $(\mathtt{e_n}', \mathtt{U_n}) = TC(\overline{\mathtt{x}}{:}\overline{\mathtt{S}}, \bullet, \mathtt{e_n})$ **in**
  let $(\mathtt{d_1}', \mathtt{V_1}) = TC(\Gamma, \mathtt{P}, \mathtt{d_1})$ **in**
    $\vdots$
  let $(\mathtt{d_m}', \mathtt{V_m}) = TC(\Gamma, \mathtt{P}, \mathtt{d_m})$ **in**
  let $\mathtt{a_1}' = \mathtt{T_1(\overline{S}\ \overline{x})\{\ return\ e_1';\ \}}$ **in**
    $\vdots$
  let $\mathtt{a_n}' = \mathtt{T_n(\overline{S}\ \overline{x})\{\ return\ e_n';\ \}}$ **in**
  if $\overline{\mathtt{V}}$ <: $\overline{\mathtt{S}} \wedge \overline{\mathtt{U}}$ <: $\overline{\mathtt{T}}$ **then** $(\mathtt{[a_1'\,,\,\cdots\,,a_n']\,(d_1'\,,\,\cdots\,,d_m)}, \mathtt{U_0})$ **else** *error*

Fig. 6. Type-checking procedure (2)

$$\frac{\mathtt{e_0} \hookrightarrow \mathtt{e_0}' \qquad \overline{\mathtt{e}} \hookrightarrow \overline{\mathtt{e}}'}{\mathtt{e_0}\langle \mathtt{T_0} \rangle\mathtt{.m(\overline{e})} \hookrightarrow \mathtt{e_0}'\langle \mathtt{T_0} \rangle\mathtt{.m(\overline{e}')}}$$

(Here, the type annotations on both sides must be the same.) The latter models type-relaxed weaving, which further allows a type annotation to be replaced with a supertype. So, the main rules are for method invocations and advice applications:

$$\frac{\mathtt{e_0} \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e_0}' \qquad \overline{\mathtt{e}} \overset{\mathrm{trw}}{\hookrightarrow} \overline{\mathtt{e}}' \qquad \mathtt{T_0} \text{ <: } \mathtt{T_0}' \qquad \mathit{mtype}(\mathtt{m}, \mathtt{T_0}') \text{ defined}}{\mathtt{e_0}\langle \mathtt{T_0} \rangle\mathtt{.m(\overline{e})} \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e_0}'\langle \mathtt{T_0}' \rangle\mathtt{.m(\overline{e}')}}$$

$$\frac{\mathtt{b} \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{b}' \qquad \overline{\mathtt{e}} \overset{\mathrm{trw}}{\hookrightarrow} \overline{\mathtt{e}}'}{\mathtt{[b]\,(\overline{e})} \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{[\overline{a},b']\,(\overline{e}')}}$$

For example, it holds that

$$\mathtt{let\ o\ =\ [b]\,()\ in\ o}\langle \mathtt{BSim} \rangle\mathtt{.show()}$$

$$
\begin{aligned}
&\hookrightarrow \mathtt{e}' \\
&= \mathtt{let\ o\ =\ [a;\ b]()\ in\ o}_{\langle \mathtt{BSim} \rangle}\mathtt{.show()}
\end{aligned}
$$

and

$$
\begin{aligned}
&\mathtt{let\ o\ =\ [b]()\ in\ o}_{\langle \mathtt{BSim} \rangle}\mathtt{.show()} \\
&\overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e}'' \\
&= \mathtt{let\ o\ =\ [a;b]()\ in\ o}_{\langle \mathtt{Tsk} \rangle}\mathtt{.show()}
\end{aligned}
$$

(where `a`, `b`, and $\mathtt{e}'$ and $\mathtt{e}''$ are from Example 1).

Note that neither relation specifies a weaving *algorithm*—how advice is chosen, in which order pieces of advice are applied, and how the base program is transformed to `[b](`$\overline{\mathtt{e}}$`)` before applying advice. They are just to model structural similarity between programs before and after (type-relaxed) weaving.

Now, correctness of the type-checking procedure is stated as follows:

Theorem 3 (Type Relaxed Weaving). *If* $\Gamma; \mathtt{P} \vdash \mathtt{e} : \mathtt{U}$ *and* $\mathtt{e} \hookrightarrow \mathtt{e}'$ *and* $TC(\Gamma, \mathtt{P}, \mathtt{e}') = (\mathtt{e}'', \mathtt{U}')$*, then* $\mathtt{e} \overset{trw}{\hookrightarrow} \mathtt{e}''$ *and* $\Gamma; \mathtt{P} \vdash \mathtt{e}'' : \mathtt{U}'$.

Proof. See Appendix C.

This theorem means that if (ordinary, type *un*changing) weaving followed by type-checking yields $\mathtt{e}''$ from a well-typed base program `e`, then $\mathtt{e}''$ is also well typed under the same type assumptions; moreover, changes in type annotations are only slight.

6 Implementation

We implemented an AspectJ compatible compiler that supports the type relaxed weaving, called *RelaxAJ*, which is publicly available[8]. The implementation is based on an existing AspectJ compiler (`ajc` version 1.6.1), and modified `ajc`'s weaving algorithm. Since the difference is only in between Rule 2 and Rule 3, we merely needed to modify a few methods in the original implementation.

The modified compiler works in the following ways. After compiling class and aspect declarations into bytecode class formats, it visits all methods in all classes provided. For each method, our weaver first accumulates all pieces of around advice applicable to any join point shadows within the method. When there are any piece of advice that violates original compiler's type-checking rule (i.e., Rule 2), our weaver performs its own type-checking based on Rule 3.

When it type-checks, the bodies of advice are woven into the bytecode instructions same as done by the original weaver. Finally, when the type-checker

[8] `http://www.graco.c.u-tokyo.ac.jp/ppp/projects/typerelaxedweaving.en`

identifies changes in target types (as discussed in Section 4.5), it changes the method invocation instructions and signatures appropriately. It also removes runtime type-checking (i.e., cast) instructions.

Our type-checker implements the typing rules presented in Section 5. In order to cope with full-set of Java bytecode instructions, which include branching ones, we implemented the algorithm as abstract interpretation.

The implementation is approximately 2,200 lines of additional code to the original AspectJ compiler. The additional type-checking adds relatively small amount of time to compilation time of a practical application program. When we compiled JHotDraw version 7.1, which consists of 441 classes or 93,000 lines of code, with a wrapper inserting aspect that has advice declarations similar to the one in Listing 1.4, the compilation time increased 2.41 percent (from 6.411 seconds to 6.566 seconds on the Sun HotSpot VM version 1.5.0 executed by two 2.8 GHz Quad-Core Intel Xeon processors with 4 GB memory, under Mac OS X version 10.5) from the case compiled by `ajc` with a dummy advice[9]. In this experiment, the advice body was woven to 51 join point shadows. Of course, the overheads would become larger when a piece of around advice that requires type relaxed weaving were woven to more methods.

7 Related Work

7.1 StrongAspectJ

StrongAspectJ is an extension to AspectJ that supports generic advice declarations in a type safe manner[7]. As mentioned in the first section, the genericity offered by StrongAspectJ is similar to, but different from the one offered by the type relaxed weaving.

Let's see similarity and difference by using the example in Listing 1.6, which is taken from (but slightly modified for explanatory purposes) StrongAspectJ's paper[7]. In Java, `Integer` and `Float` are both subtypes of the `Number` interface, which requires the `intValue` method.

Interestingly, the advice declaration, which is rejected by AspectJ, is accepted both by StrongAspectJ (modulo slight modifications to the advice declaration) and the relaxed type weaving. With StrongAspectJ, we can redefine the advice by using a type variable, so that the type system can check correctness of the advice body with respect to *the type of each join point* where the advice is applied to. With the type relaxed weaving, the advice happens to be accepted because *the return values from the join points are used* merely as the `Number` objects.

Difference becomes apparent when we modify either the advice or the `compute` method. If we modify the body of the advice so that it returns, for example,

[9] We declared the return type of the advice as `Object` in order to get the advice compiled by AspectJ. With this advice, `ajc` can generate woven code, which cause runtime cast errors.

```
Integer calcInteger() { return new Integer(...); }
Float   calcFloat()   { return new Float  (...); }

int compute() {
  Integer i = calcInteger();
  Float   f = calcFloat  ();
  return i.intValue() + f.intValue();
}

Number around():call((Integer||Float) calc*(..)) {
  Number n = proceed();
  while (n.intValue() > 100)
    n = proceed();
  return n;
}
```

Listing 1.6. An around advice declaration that is applied to join points of different types.

`new Double(0)`, StrongAspectJ cannot accept such a piece of advice. Alternatively, if we modify the `compute` method so that it calls a method that is not defined in `Integer` class on `f`, the type relaxed weaving cannot accept the advice any longer.

We believe that the type systems of StrongAspectJ and the type relaxed weaving complement each other, and are currently designing an AspectJ language extension that supports both mechanisms.

7.2 Type Systems for Around Advice

As far as the authors know, all formalizations of around advice are based on Rule 2; i.e., advice types are checked against types in join points, if they formalized type systems. Wand, Kiczales and Dutchyn formalized behavior of around advice without types[20]. Clifton and Leavens formalized the proceed mechanism of around advice and its type safety[3]. Their type system is based on the one in AspectJ, which is based on Rule 2.

AspectML[5] and Aspectual Caml[16] are AOP extensions to functional languages. Those languages support *polymorphic advice*, which can be applied to join points that have polymorphic types. Although polymorphic types might give similar genericity, we believe that our work first pointed out the problem in practice, and formalized in a language with a subtyping relation.

8 Conclusion

This paper presented the type relaxed weaving, a novel weaving mechanism for around advice in statically typed AOP languages. With the type relaxed weaving,

we can define around advice that replaces a value in a join point with the one of a different type, as long as the usage of the value in subsequent computation agrees.

Our contributions are: we (1) pointed out that the problem of the type-checking rules on around advice in existing AOP languages, (2) proposed the type relaxed weaving, which resolves the problem in a type safe manner, (3) formalized the core part of the mechanism in order to show type soundness, and (4) implemented the weaving mechanism as an AspectJ compatible compiler, which is publicly available.

We are extending our compiler so that it will allow around advice to provide values of different types to `proceed()`, as it allows to provide values to return from the advice. We are also designing a language *StrongRelaxAJ*, which is a hybrid of StrongAspectJ and RelaxAJ by taking advantages of both mechanisms[1].

References

1. Aotani, T., Toyama, M., Masuhara, H.: StrongRelaxAJ: integrating adaptability of RelaxAJ and expressiveness of StrongAspectJ. In: Ostermann, K. (ed.) Foundations of Aspect-Oriented Langauges (FOAL2010). pp. 1–4 (Mar 2010), `http://www.eecs.ucf.edu/FOAL/papers-2010/proceedings.pdf`, technical report CS-TR-10-04, School of Electrical Engineering and Computer Science, University of Central Florida
2. Bodkin, R.: Performance monitoring with AspectJ: A look inside the Glassbox inspector with AspectJ and JMX. AOP@Work (Sep 2005), `http://www-128.ibm.com/developerworks/java/library/j-aopwork10/`
3. Clifton, C., Leavens, G.T.: MiniMAO1: Investigating the semantics of proceed. Science of Computer Programming 63(3), 321–374 (Dec 2006), preprint: Department of Computer Science, Iowa State University, TR #05-01, January 2005
4. Colyer, A., Clement, A.: Large-scale AOSD for middleware. In: Lieberherr, K. (ed.) Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04). pp. 56–65. ACM Press, New York, NY, USA (Mar 2004)
5. Dantas, D.S., Walker, D., Washburn, G., Weirich, S.: AspectML: A polymorphic aspect-oriented functional programming language. Transactions on Programming Languages and Systems 30(3), 1–60 (2008), `http://www.cs.princeton.edu/sip/projects/aspectml/`
6. Erwig, M., Ren, D.: Type-safe update programming. In: ESOP 2003. Lecture Notes in Computer Science, vol. 2618, pp. 269–283 (2003)
7. Fraine, B.D., Südholt, M., Jonckers, V.: StrongAspectJ: flexible and safe pointcut/advice bindings. In: Mezini, M. (ed.) Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08). pp. 60–71. ACM Press, New York, NY, USA (Apr 2008)
8. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Prentice Hall, third edn. (Jun 2005)
9. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. In: OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 132–146. ACM, New York, NY, USA (1999)

10. Igarashi, A., Nagira, H.: Union types for object-oriented programming. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing. pp. 1435–1441. ACM, New York, NY, USA (2006)
11. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP 2001). Lecture Notes in Computer Science, vol. 2072, pp. 327–353. Springer-Verlag (Jun 2001), http://www.parc.xerox.com/groups/csl/projects/aspectj/downloads/ECOOP2001-Overview.pdf
12. Knudsen, J.L.: Name collision in multiple classification hierarchies. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 322, pp. 93–109. Springer-Verlag, London, UK (1988)
13. Lämmel, R.: A semantical approach to method-call interception. In: Kiczales, G. (ed.) Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02). pp. 41–55. ACM Press (Apr 2002)
14. Leroy, X.: Java bytecode verification: algorithms and formalizations. Journal of Automated Reasoning 30(3-4), 235–269 (2003)
15. Masuhara, H., Igarashi, A., Toyama, M.: Type relaxed weaving. In: Südholt, M. (ed.) Proceedings of the 9th International Conference on Aspect-Oriented Software Development (10). pp. 121–132. ACM Press, New York, NY, USA (18 Mar 2010)
16. Masuhara, H., Tatsuzawa, H., Yonezawa, A.: Aspectual Caml: an aspect-oriented functional language. In: Pierce, B. (ed.) Proceedings of International Conference on Functional Programming (ICFP 2005). pp. 320–330 (Sep 2005)
17. Mezini, M., Ostermann, K.: Conquering aspects with Caesar. In: Akşit, M. (ed.) Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03). pp. 90–99. ACM Press (Mar 2003)
18. Rashid, A., Chitchyan, R.: Persistence as an aspect. In: Akşit, M. (ed.) Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03). pp. 120–129. ACM Press (Mar 2003)
19. Spinczyk, O., Gal, A., Schroder-Preikschat, W.: AspectC++: An aspect-oriented extension to C++. In: Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002). pp. 18–21. Sydney, Australia (Feb 2002), http://www.aspectc.org/download/tools2002.ps.gz
20. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join points in aspect-oriented programming. ACM Transactions on Programming Languages and Systems (TOPLAS) 26(5), 890–910 (Sep 2004), wkd02.ps, earlier versions of this paper were presented at the 9th International Workshop on Foundations of Object-Oriented Languages, January 19, 2002, and at the Workshop on Foundations of Aspect-Oriented Languages (FOAL), April 22, 2002.
21. Wiese, D., Meunier, R., Hohenstein, U.: How to convince industry of AOP. In: Proceedings of Industry Track at AOSD.07 (Mar 2007), http://aosd.net/2007/program/industry/
22. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (Nov 1994)

A Complete Definition of FJR

A.1 Syntax

$$
\begin{array}{rcl}
\texttt{CL} & ::= & \texttt{class C extends C implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} \\
\texttt{M} & ::= & \texttt{T m(}\overline{\texttt{T x}}\texttt{)\{ return e; \}} \\
\texttt{IF} & ::= & \texttt{interface I \{ } \overline{\mathcal{S}} \texttt{ \}} \\
\mathcal{S} & ::= & \texttt{T m(}\overline{\texttt{T x}}\texttt{);} \\
\texttt{a}, \texttt{b} & ::= & \texttt{T(}\overline{\texttt{T x}}\texttt{)\{ return e; \}} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e}_{\langle \texttt{T} \rangle}\texttt{.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C()} \mid \texttt{let x = e in e} \\
 & & \mid \texttt{(?e:e)} \mid \texttt{proceed(}\overline{\texttt{e}}\texttt{)} \mid \texttt{[}\overline{\texttt{a}}\texttt{](}\overline{\texttt{e}}\texttt{)} \\
\texttt{S}, \texttt{T} & ::= & \texttt{C} \mid \texttt{I} \\
\texttt{U}, \texttt{V} & ::= & \texttt{T} \mid \texttt{U} \cup \texttt{U} \\
\texttt{P} & ::= & \bullet \mid \overline{\texttt{T}} \rightarrow \texttt{U}
\end{array}
$$

A.2 Lookup Functions

$\boxed{\mathit{mtype}(\texttt{m}, \texttt{T}) = \overline{\texttt{T}} \rightarrow \texttt{T}_0}$

$$
\frac{\begin{array}{c}\texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} \\ \texttt{T}_0 \texttt{ m(}\overline{\texttt{T x}}\texttt{)\{ return e; \}} \in \overline{\texttt{M}}\end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{T}} \rightarrow \texttt{T}_0} \quad \text{(MT-Class)}
$$

$$
\frac{\texttt{interface I \{ } \overline{\mathcal{S}} \texttt{ \}} \qquad \texttt{T}_0 \texttt{ m(}\overline{\texttt{T x}}\texttt{);} \in \overline{\mathcal{S}}}{\mathit{mtype}(\texttt{m}, \texttt{I}) = \overline{\texttt{T}} \rightarrow \texttt{T}_0} \quad \text{(MT-Interface)}
$$

$$
\frac{\texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{M}} \qquad \mathit{mtype}(\texttt{m}, \texttt{D}) = \overline{\texttt{T}} \rightarrow \texttt{T}_0}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{T}} \rightarrow \texttt{T}_0} \quad \text{(MT-SuperC)}
$$

$$
\frac{\texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{M}} \qquad \mathit{mtype}(\texttt{m}, \texttt{I}_i) = \overline{\texttt{T}} \rightarrow \texttt{T}_0}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{T}} \rightarrow \texttt{T}_0} \quad \text{(MT-SuperI)}
$$

$\boxed{\mathit{mtype}^C(\texttt{m}, \texttt{C}) = \overline{\texttt{T}} \rightarrow \texttt{T}_0}$

$$
\frac{\begin{array}{c}\texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} \\ \texttt{T}_0 \texttt{ m(}\overline{\texttt{T x}}\texttt{)\{ return e; \}} \in \overline{\texttt{M}}\end{array}}{\mathit{mtype}^C(\texttt{m}, \texttt{C}) = \overline{\texttt{T}} \rightarrow \texttt{T}_0} \quad \text{(MTC-Class)}
$$

$$\frac{\texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{M}} \qquad \mathit{mtype}^C(\texttt{m}, \texttt{D}) = \overline{\texttt{T}} \to \texttt{T}_0}{\mathit{mtype}^C(\texttt{m}, \texttt{C}) = \overline{\texttt{T}} \to \texttt{T}_0} \quad \text{(MTC-Super)}$$

$\boxed{\mathit{mbody}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.\texttt{e}}$

$$\frac{\texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} \qquad \texttt{T}_0\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ return e; \}} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.\texttt{e}} \quad \text{(MB-Class)}$$

$$\frac{\texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ \{ } \overline{\texttt{M}} \texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{M}} \qquad \mathit{mbody}(\texttt{m}, \texttt{D}) = \overline{\texttt{x}}.\texttt{e}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.\texttt{e}} \quad \text{(MB-Super)}$$

$\boxed{\mathit{rettype}(\texttt{a}) = \texttt{T}}$

$$\mathit{rettype}(\texttt{T(}\overline{\texttt{S}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}}) = \texttt{T}$$

A.3 Subtyping

$\boxed{\texttt{U <: V}}$

$$\texttt{U} <: \texttt{U} \quad \text{(S-Refl)}$$

$$\frac{\texttt{U}_1 <: \texttt{U}_2 \qquad \texttt{U}_2 <: \texttt{U}_3}{\texttt{U}_1 <: \texttt{U}_3} \quad \text{(S-Trans)}$$

$$\texttt{U} <: \texttt{Object} \quad \text{(S-Object)}$$

$$\frac{\texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ \{} \cdots \texttt{\}}}{\texttt{C} <: \texttt{D}} \quad \text{(S-Extends)}$$

$$\frac{\texttt{class C extends D implements } \overline{\texttt{I}} \texttt{ \{} \cdots \texttt{\}}}{\texttt{C} <: \texttt{I}_i} \quad \text{(S-Implements)}$$

$$\texttt{U}_1 <: \texttt{U}_1 \cup \texttt{U}_2 \quad \text{(S-UnionR1)}$$

$$\texttt{U}_2 <: \texttt{U}_1 \cup \texttt{U}_2 \quad \text{(S-UnionR2)}$$

$$\frac{\texttt{U}_1 <: \texttt{U}_3 \qquad \texttt{U}_2 <: \texttt{U}_3}{\texttt{U}_1 \cup \texttt{U}_2 <: \texttt{U}_3} \quad \text{(S-UnionL)}$$

A.4 Typing

$\boxed{\Gamma;\mathtt{P} \vdash \mathtt{e} : \mathtt{U}}$

$$\Gamma;\mathtt{P} \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-Var)}$$

$$\frac{\Gamma;\mathtt{P} \vdash \mathtt{e}_0 : \mathtt{U}_0 \qquad \mathtt{U}_0 \mathrel{<:} \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{T}} \to \mathtt{T} \qquad \Gamma;\mathtt{P} \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \overline{\mathtt{U}} \mathrel{<:} \overline{\mathtt{T}}}{\Gamma;\mathtt{P} \vdash \mathtt{e}_{0\langle \mathtt{T}_0 \rangle}.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{T}} \qquad \text{(T-Invk)}$$

$$\Gamma;\mathtt{P} \vdash \mathtt{new\ C()} : \mathtt{C} \qquad \text{(T-New)}$$

$$\frac{\Gamma;\mathtt{P} \vdash \mathtt{e}_1 : \mathtt{U}_1 \qquad \Gamma, \mathtt{x}{:}\mathtt{U}_1;\mathtt{P} \vdash \mathtt{e}_2 : \mathtt{U}_2}{\Gamma;\mathtt{P} \vdash \mathtt{let\ x = e_1\ in\ e_2} : \mathtt{U}_2} \qquad \text{(T-Let)}$$

$$\frac{\Gamma;\mathtt{P} \vdash \mathtt{e}_1 : \mathtt{U}_1 \qquad \Gamma;\mathtt{P} \vdash \mathtt{e}_2 : \mathtt{U}_2}{\Gamma;\mathtt{P} \vdash \mathtt{(?e_1 : e_2)} : \mathtt{U}_1 \cup \mathtt{U}_2} \qquad \text{(T-Choice)}$$

$$\frac{\Gamma;\overline{\mathtt{T}} \to \mathtt{U} \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \qquad \overline{\mathtt{V}} \mathrel{<:} \overline{\mathtt{T}}}{\Gamma;\overline{\mathtt{T}} \to \mathtt{U} \vdash \mathtt{proceed}(\overline{\mathtt{e}}) : \mathtt{U}} \qquad \text{(T-Proceed)}$$

$$\frac{\overline{\mathtt{S}} \to \mathtt{U}_0 \vdash \overline{\mathtt{a}}\ \mathtt{OK} \qquad \bullet \vdash \mathtt{b}\ \mathtt{OK} \qquad \mathtt{U}_0 = \bigcup_{\mathtt{a} \in \overline{\mathtt{a}}, \mathtt{b}} \mathit{rettype}(\mathtt{a}) \qquad \Gamma;\mathtt{P} \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \qquad \overline{\mathtt{V}} \mathrel{<:} \overline{\mathtt{S}}}{\Gamma;\mathtt{P} \vdash \mathtt{[}\overline{\mathtt{a}}\mathtt{,b]}(\overline{\mathtt{e}}) : \mathtt{U}_0} \qquad \text{(T-Woven)}$$

$\boxed{\mathtt{P} \vdash \mathtt{a}\ \mathtt{OK}}$

$$\frac{\mathtt{P} = \bullet \text{ or } \overline{\mathtt{S}} \to \mathtt{V} \qquad \overline{\mathtt{x}}{:}\overline{\mathtt{S}};\mathtt{P} \vdash \mathtt{e} : \mathtt{U} \qquad \mathtt{U} \mathrel{<:} \mathtt{T}}{\mathtt{P} \vdash \mathtt{T}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}\ \mathtt{OK}} \qquad \text{(T-Advice)}$$

$\boxed{\mathtt{M}\ \mathtt{OK\ IN}\ \mathtt{C}}$

$$\frac{\overline{\mathtt{x}}{:}\overline{\mathtt{T}}, \mathtt{this}{:}\mathtt{C};\bullet \vdash \mathtt{e} : \mathtt{U} \qquad \mathtt{U} \mathrel{<:} \mathtt{T}_0 \qquad \mathtt{class\ C\ extends\ D\ implements}\ \overline{\mathtt{I}}\ \{\cdots\} \qquad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{T}} \to \mathtt{T}_0)}{\mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}\ \mathtt{OK\ IN\ C}} \qquad \text{(T-Method)}$$

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

$\boxed{\mathtt{C}\ \mathtt{OK}}$

$$\frac{\overline{\mathtt{M}}\ \mathtt{OK\ IN\ C} \qquad \forall \mathtt{m}, \mathtt{I} \in \overline{\mathtt{I}}.(\mathit{mtype}(\mathtt{m}, \mathtt{I}) = \overline{\mathtt{T}} \to \mathtt{T}_0) \Longrightarrow (\mathit{mtype}^C(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{T}_0)}{\mathtt{class\ C\ extends\ D\ implements}\ \overline{\mathtt{I}}\ \{\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \qquad \text{(T-Class)}$$

A.5 Operational Semantics

$\boxed{\mathtt{e} \longrightarrow \mathtt{e}'}$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0}{\mathtt{new\ C()}_{\langle \mathtt{T} \rangle}\mathtt{.m}(\overline{\mathtt{e}}) \longrightarrow [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ C()/this}]\mathtt{e}_0} \quad \text{(R-Invk)}$$

$$\mathtt{let\ x\ =\ e}_1\ \mathtt{in\ e}_2 \longrightarrow [\mathtt{e}_1/\mathtt{x}]\mathtt{e}_2 \quad \text{(R-Let)}$$

$$(?\mathtt{e}_1 : \mathtt{e}_2) \longrightarrow \mathtt{e}_i \quad \text{(R-Choice)}$$

$$\frac{\mathtt{a}_0 = \mathtt{T}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}_0\mathtt{;}\ \}}{[\mathtt{a}_0, \overline{\mathtt{a}}](\overline{\mathtt{e}}) \longrightarrow [\overline{\mathtt{e}}/\overline{\mathtt{x}}, [\overline{\mathtt{a}}]/\mathtt{proceed}]\mathtt{e}_0} \quad \text{(R-Advice)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_{0\langle \mathtt{T} \rangle}\mathtt{.m}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0{}'_{\langle \mathtt{T} \rangle}\mathtt{.m}(\overline{\mathtt{e}})} \quad \text{(RC-InvkRecv)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{e}_{0\langle \mathtt{T} \rangle}\mathtt{.m}(\ldots, \mathtt{e}_i, \ldots) \longrightarrow \mathtt{e}_{0\langle \mathtt{T} \rangle}\mathtt{.m}(\ldots, \mathtt{e}_i{}', \ldots)} \quad \text{(RC-InvkArg)}$$

$$\frac{\mathtt{e}_1 \longrightarrow \mathtt{e}_1{}'}{\mathtt{let\ x\ =\ e}_1\ \mathtt{in\ e}_2 \longrightarrow \mathtt{let\ x\ =\ e}_1{}'\ \mathtt{in\ e}_2} \quad \text{(RC-Let1)}$$

$$\frac{\mathtt{e}_2 \longrightarrow \mathtt{e}_2{}'}{\mathtt{let\ x\ =\ e}_1\ \mathtt{in\ e}_2 \longrightarrow \mathtt{let\ x\ =\ e}_1\ \mathtt{in\ e}_2{}'} \quad \text{(RC-Let2)}$$

$$\frac{\mathtt{e}_1 \longrightarrow \mathtt{e}_1{}'}{(?\mathtt{e}_1 : \mathtt{e}_2) \longrightarrow (?\mathtt{e}_1{}' : \mathtt{e}_2)} \quad \text{(RC-Choice1)}$$

$$\frac{\mathtt{e}_2 \longrightarrow \mathtt{e}_2{}'}{(?\mathtt{e}_1 : \mathtt{e}_2) \longrightarrow (?\mathtt{e}_1 : \mathtt{e}_2{}')} \quad \text{(RC-Choice2)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{[\overline{\mathtt{a}}](\cdots, \mathtt{e}_i, \cdots) \longrightarrow [\overline{\mathtt{a}}](\cdots, \mathtt{e}_i{}', \cdots)} \quad \text{(RC-WovenArg)}$$

B Type-Checking Procedure

$\boxed{TC(\Gamma, \mathtt{P}, \mathtt{e}) = (\mathtt{e}', \mathtt{U})}$

$TC(\Gamma, \mathtt{P}, \mathtt{x}) = (\mathtt{x}, \Gamma(\mathtt{x}))$

$TC(\Gamma, \mathtt{P}, \mathtt{let\ x\ =\ e_1\ in\ e_2}) =$
let $(\mathtt{e_1}', \mathtt{U_1}) = TC(\Gamma, \mathtt{P}, \mathtt{e_1})$ **in**
let $(\mathtt{e_2}', \mathtt{U_2}) = TC((\Gamma, \mathtt{x{:}U_1}), \mathtt{P}, \mathtt{e_2})$ **in**
$(\mathtt{let\ x\ =\ e_1}'\ \mathtt{in\ e_2}', \mathtt{U_2})$

$TC(\Gamma, \mathtt{P}, \mathtt{e}_{0\langle \mathtt{T} \rangle}\mathtt{.m(e_1\,,\ \cdots\,,e_n)}) =$
let $(\mathtt{e_0}', \mathtt{U_0}) = TC(\Gamma, \mathtt{P}, \mathtt{e_0})$ **in**
let $(\mathtt{e_1}', \mathtt{U_1}) = TC(\Gamma, \mathtt{P}, \mathtt{e_1})$ **in**
$\vdots$
let $(\mathtt{e_n}', \mathtt{U_n}) = TC(\Gamma, \mathtt{P}, \mathtt{e_n})$ **in**
let $\overline{\mathtt{T}}{\to}\mathtt{T_0} = \mathit{mtype}(\mathtt{m}, \mathtt{T})$ **in**
let S be a simple type such that $\mathtt{U_0} \cup \mathtt{T} <: \mathtt{S}$ and $\mathit{mtype}(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{T}}{\to}\mathtt{T_0}$ **in**
if $\overline{\mathtt{U}} <: \overline{\mathtt{T}}$ **then** $(\mathtt{e_0}'_{\langle \mathtt{S} \rangle}\mathtt{.m(e_1'\,,\ \cdots\,,e_n')}, \mathtt{T_0})$ **else** *error*

$TC(\Gamma, \mathtt{P}, \mathtt{new\ C()}) = (\mathtt{new\ C()}, \mathtt{C})$

$TC(\Gamma, \mathtt{P}, \mathtt{(?e_1{:}e_2)}) =$
let $(\mathtt{e_1}', \mathtt{U_1}) = TC(\Gamma, \mathtt{P}, \mathtt{e_1})$ **in**
let $(\mathtt{e_2}', \mathtt{U_2}) = TC(\Gamma, \mathtt{P}, \mathtt{e_2})$ **in**
$(\mathtt{(?e_1'{:}e_2')}, \mathtt{U_1} \cup \mathtt{U_2})$

$TC(\Gamma, \overline{\mathtt{S}}{\to}\mathtt{U}, \mathtt{proceed(e_1\,,\ \cdots\,,e_n)}) =$
let $(\mathtt{e_1}', \mathtt{U_1}) = TC(\Gamma, \overline{\mathtt{S}}{\to}\mathtt{U}, \mathtt{e_1})$ **in**
$\vdots$
let $(\mathtt{e_n}', \mathtt{U_n}) = TC(\Gamma, \overline{\mathtt{S}}{\to}\mathtt{U}, \mathtt{e_n})$ **in**
if $\overline{\mathtt{U}} <: \overline{\mathtt{S}}$ **then** $(\mathtt{proceed(e_1'\,,\ \cdots\,,e_n')}, \mathtt{U})$ **else** *error*

$TC(\Gamma, \mathtt{P}, \mathtt{[a_1\,,\ \cdots\,,a_n](d_1\,,\ \cdots\,,d_m)}) =$
let $\mathtt{T_1(\overline{S}\ \overline{x})\{\ return\ e_1;\ \}} = \mathtt{a_1}$ **in**
$\vdots$
let $\mathtt{T_n(\overline{S}\ \overline{x})\{\ return\ e_n;\ \}} = \mathtt{a_n}$ **in**
let $\mathtt{U_0} = \mathtt{T_1} \cup \cdots \cup \mathtt{T_n}$ **in**
let $(\mathtt{e_1}', \mathtt{U_1}) = TC(\overline{\mathtt{x}}{:}\overline{\mathtt{S}}, \overline{\mathtt{S}}{\to}\mathtt{U_0}, \mathtt{e_1})$ **in**
$\vdots$
let $(\mathtt{e_{n-1}}', \mathtt{U_{n-1}}) = TC(\overline{\mathtt{x}}{:}\overline{\mathtt{S}}, \overline{\mathtt{S}}{\to}\mathtt{U_0}, \mathtt{e_{n-1}})$ **in**
let $(\mathtt{e_n}', \mathtt{U_n}) = TC(\overline{\mathtt{x}}{:}\overline{\mathtt{S}}, \bullet, \mathtt{e_n})$ **in**
let $(\mathtt{d_1}', \mathtt{V_1}) = TC(\Gamma, \mathtt{P}, \mathtt{d_1})$ **in**

$\vdots$

let $(\mathtt{d}_m{}', \mathtt{V}_m) = TC(\Gamma, \mathtt{P}, \mathtt{d}_m)$ **in**
let $\mathtt{a}_1{}' = \mathtt{T}_1(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_1{}';\ \}$ **in**

$\vdots$

let $\mathtt{a}_n{}' = \mathtt{T}_n(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_n{}';\ \}$ **in**
if $\overline{\mathtt{V}} <: \overline{\mathtt{S}} \wedge \overline{\mathtt{U}} <: \overline{\mathtt{T}}$ **then** $([\mathtt{a}_1{}', \cdots, \mathtt{a}_n{}'](\mathtt{d}_1{}', \cdots, \mathtt{d}_m{}'), \mathtt{U}_0)$ **else** *error*

$\boxed{\mathtt{e} \hookrightarrow \mathtt{e}'}$

$$\mathtt{x} \hookrightarrow \mathtt{x}'$$

$$\frac{\mathtt{e}_1 \hookrightarrow \mathtt{e}_1{}' \qquad \mathtt{e}_2 \hookrightarrow \mathtt{e}_2{}'}{\mathtt{let\ x\ =\ e}_1\ \mathtt{in\ e}_2 \hookrightarrow \mathtt{let\ x\ =\ e}_1{}'\ \mathtt{in\ e}_2{}'}$$

$$\frac{\mathtt{e}_0 \hookrightarrow \mathtt{e}_0{}' \qquad \overline{\mathtt{e}} \hookrightarrow \overline{\mathtt{e}}'}{\mathtt{e}_{0\langle \mathtt{T}_0 \rangle}.\mathtt{m}(\overline{\mathtt{e}}) \hookrightarrow \mathtt{e}_0{}'_{\langle \mathtt{T}_0 \rangle}.\mathtt{m}(\overline{\mathtt{e}}')}$$

$$\mathtt{new\ C()} \hookrightarrow \mathtt{new\ C()}$$

$$\frac{\mathtt{e}_1 \hookrightarrow \mathtt{e}_1{}' \qquad \mathtt{e}_2 \hookrightarrow \mathtt{e}_2{}'}{(\mathtt{?e}_1\mathtt{:e}_2) \hookrightarrow (\mathtt{?e}_1{}'\mathtt{:e}_2{}')}$$

$$\frac{\overline{\mathtt{e}} \hookrightarrow \overline{\mathtt{e}}'}{\mathtt{proceed}(\overline{\mathtt{e}}) \hookrightarrow \mathtt{proceed}(\overline{\mathtt{e}}')}$$

$$\frac{\overline{\mathtt{b}} \hookrightarrow \overline{\mathtt{b}}' \qquad \overline{\mathtt{e}} \hookrightarrow \overline{\mathtt{e}}'}{[\mathtt{b}](\overline{\mathtt{e}}) \hookrightarrow [\overline{\mathtt{a}}, \mathtt{b}'](\overline{\mathtt{e}}')}$$

$$\frac{\mathtt{e} \hookrightarrow \mathtt{e}'}{\mathtt{T}_0(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \} \hookrightarrow \mathtt{T}_0(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}';\ \}}$$

$\boxed{\mathtt{e} \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e}'}$

$$\mathtt{x} \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{x}'$$

$$\frac{\mathtt{e}_1 \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e}_1{}' \qquad \mathtt{e}_2 \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e}_2{}'}{\mathtt{let\ x\ =\ e}_1\ \mathtt{in\ e}_2 \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{let\ x\ =\ e}_1{}'\ \mathtt{in\ e}_2{}'}$$

$$\frac{\mathtt{e}_0 \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e}_0{}' \qquad \overline{\mathtt{e}} \overset{\mathrm{trw}}{\hookrightarrow} \overline{\mathtt{e}}' \qquad \mathtt{T}_0 <: \mathtt{T}_0{}' \qquad \mathit{mtype}(\mathtt{m}, \mathtt{T}_0{}') \text{ defined}}{\mathtt{e}_{0\langle \mathtt{T}_0 \rangle}.\mathtt{m}(\overline{\mathtt{e}}) \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e}_0{}'_{\langle \mathtt{T}_0{}' \rangle}.\mathtt{m}(\overline{\mathtt{e}}')}$$

$$\texttt{new C()} \overset{\text{trw}}{\hookrightarrow} \texttt{new C()}$$

$$\frac{\texttt{e}_1 \overset{\text{trw}}{\hookrightarrow} \texttt{e}_1' \qquad \texttt{e}_2 \overset{\text{trw}}{\hookrightarrow} \texttt{e}_2'}{(\texttt{?e}_1\texttt{:e}_2) \overset{\text{trw}}{\hookrightarrow} (\texttt{?e}_1'\texttt{:e}_2')}$$

$$\frac{\overline{\texttt{e}} \overset{\text{trw}}{\hookrightarrow} \overline{\texttt{e}}'}{\texttt{proceed}(\overline{\texttt{e}}) \overset{\text{trw}}{\hookrightarrow} \texttt{proceed}(\overline{\texttt{e}}')}$$

$$\frac{\texttt{b} \overset{\text{trw}}{\hookrightarrow} \texttt{b}' \qquad \overline{\texttt{e}} \overset{\text{trw}}{\hookrightarrow} \overline{\texttt{e}}'}{\texttt{[b]}(\overline{\texttt{e}}) \overset{\text{trw}}{\hookrightarrow} \texttt{[}\overline{\texttt{a}}\texttt{,b']}(\overline{\texttt{e}}')}$$

$$\frac{\texttt{e} \overset{\text{trw}}{\hookrightarrow} \texttt{e}'}{\texttt{T}_0\texttt{(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \overset{\text{trw}}{\hookrightarrow} \texttt{T}_0\texttt{(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e'; \}}}$$

C Proofs

C.1 Proof of Theorems 1 and 2

Lemma 1 (Weakening).

1. *If* $\Gamma; \texttt{P} \vdash \texttt{e} : \texttt{U}$*, then* $\Gamma, \texttt{x} : \texttt{V}; \texttt{P} \vdash \texttt{e} : \texttt{U}$ *for any* $\texttt{V}$ *and* $\texttt{x} \notin dom(\Gamma)$.
2. *If* $\Gamma; \bullet \vdash \texttt{e} : \texttt{U}$*, then* $\Gamma; \overline{\texttt{S}}\texttt{→}\texttt{V} \vdash \texttt{e} : \texttt{U}$ *for any* $\overline{\texttt{S}}$ *and* $\texttt{V}$.

Proof. By easy induction on $\texttt{e}$.

Lemma 2 (Narrowing).

1. *If* $\Gamma, \texttt{x} : \texttt{V}; \texttt{P} \vdash \texttt{e} : \texttt{U}$ *and* $\texttt{V}' \texttt{ <: } \texttt{V}$*, then there exists* $\texttt{U}'$ *such that* $\Gamma, \texttt{x:V}'; \texttt{P} \vdash \texttt{e} : \texttt{U}'$ *and* $\texttt{U}' \texttt{ <: } \texttt{U}$.
2. *If* $\Gamma; \overline{\texttt{S}}\texttt{→}\texttt{U}_1 \vdash \texttt{e} : \texttt{U}$ *and* $\texttt{U}_2 \texttt{ <: } \texttt{U}_1$*, then there exists* $\texttt{U}'$ *such that* $\Gamma; \overline{\texttt{S}}\texttt{→}\texttt{U}_2 \vdash \texttt{e} : \texttt{U}'$ *and* $\texttt{U}' \texttt{ <: } \texttt{U}$.

Proof. By easy induction on $\texttt{e}$.

Lemma 3 (Substitution Lemma). *If* $\Gamma, \overline{\texttt{x}}\texttt{:}\overline{\texttt{U}}; \texttt{P} \vdash \texttt{e} : \texttt{U}_0$ *and* $\Gamma; \texttt{P} \vdash \overline{\texttt{e}} : \overline{\texttt{V}}$ *and* $\overline{\texttt{V}} \texttt{ <: } \overline{\texttt{U}}$*, then there exists some type* $\texttt{V}_0$ *such that* $\Gamma; \texttt{P} \vdash [\overline{\texttt{e}}/\overline{\texttt{x}}]\texttt{e} : \texttt{V}_0$ *and* $\texttt{V}_0 \texttt{ <: } \texttt{U}_0$.

Proof. By induction on $\texttt{e}$ with case analysis on the last typing rule used. We show only interesting cases.

Case T-Invk**:** We have

$$\begin{array}{lll} \texttt{e} = \texttt{e}_0\texttt{.m}(\overline{\texttt{d}}) & \Gamma, \overline{\texttt{x}}\texttt{:}\overline{\texttt{U}}; \texttt{P} \vdash \texttt{e}_0 : \texttt{U}_{00} & \texttt{U}_{00} \texttt{ <: } \texttt{T} \\ \mathit{mtype}(\texttt{m}, \texttt{T}) = \overline{\texttt{T}}\texttt{→}\texttt{T}_0 & \Gamma, \overline{\texttt{x}}\texttt{:}\overline{\texttt{U}}; \texttt{P} \vdash \overline{\texttt{d}} : \overline{\texttt{V}} & \overline{\texttt{V}} \texttt{ <: } \overline{\texttt{T}} \end{array}$$

for some $\mathtt{e}_0$, $\overline{\mathtt{d}}$, $\mathtt{m}$, $\mathtt{T}$, $\overline{\mathtt{T}}$, and $\overline{\mathtt{V}}$ and $\mathtt{T}_0 = \mathtt{U}_0$. By the induction hypothesis, there exist some $\mathtt{V}_{00}$ and $\overline{\mathtt{V}}'$ such that

$$\begin{array}{ll} \Gamma; \mathtt{P} \vdash [\overline{\mathtt{e}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{V}_{00} & \mathtt{V}_{00} \mathrel{<:} \mathtt{U}_{00} \\ \Gamma; \mathtt{P} \vdash [\overline{\mathtt{e}}/\overline{\mathtt{x}}]\overline{\mathtt{d}} : \overline{\mathtt{V}}' & \overline{\mathtt{V}}' \mathrel{<:} \overline{\mathtt{V}} \end{array}$$

By S-Trans, $\mathtt{V}_{00} \mathrel{<:} \mathtt{T}$ and $\overline{\mathtt{V}}' \mathrel{<:} \overline{\mathtt{T}}$. By T-Invk, we have $\Gamma; \mathtt{P} \vdash [\overline{\mathtt{e}}/\overline{\mathtt{x}}](\mathtt{e}_0\,.\mathtt{m}(\overline{\mathtt{e}})) : \mathtt{T}_0$.

Case T-Let**:** We have

$$\begin{array}{l} \mathtt{e} = \mathtt{let\ y\ =\ e}_1\ \mathtt{in\ e}_2 \\ \Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{U}}; \mathtt{P} \vdash \mathtt{e}_1 : \mathtt{U}_y \qquad \Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{U}}, \mathtt{y}{:}\mathtt{U}_y; \mathtt{P} \vdash \mathtt{e}_1 : \mathtt{U}_0 \end{array}$$

By the induction hypothesis, there exists some $\mathtt{U}_y{}'$ such that

$$\Gamma; \mathtt{P} \vdash [\overline{\mathtt{e}}/\overline{\mathtt{x}}]\mathtt{e}_1 : \mathtt{U}_y{}' \qquad \mathtt{U}_y{}' \mathrel{<:} \mathtt{U}_y.$$

By Lemma 2,

$$\Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{U}}, \mathtt{y}{:}\mathtt{U}_y{}'; \mathtt{P} \vdash \mathtt{e}_2 : \mathtt{U}_0{}' \qquad \mathtt{U}_0{}' \mathrel{<:} \mathtt{U}_0$$

for some $\mathtt{U}_0{}'$. Again, by the induction hypothesis,

$$\Gamma, \mathtt{y}{:}\mathtt{U}_y{}'; \mathtt{P} \vdash [\overline{\mathtt{e}}/\overline{\mathtt{x}}]\mathtt{e}_2 : \mathtt{U}_0{}'' \qquad \mathtt{U}_0{}'' \mathrel{<:} \mathtt{U}_0{}'$$

for some $\mathtt{U}_0{}''$. By T-Let and S-Trans,

$$\Gamma; \mathtt{P} \vdash [\overline{\mathtt{e}}/\overline{\mathtt{x}}](\mathtt{let\ y\ =\ e}_1\ \mathtt{in\ e}_2) : \mathtt{U}_0{}'' \qquad \mathtt{U}_0{}'' \mathrel{<:} \mathtt{U}_0.$$

The other cases are easy.

Lemma 4 (Advice Substitution). *If* $\Gamma; \overline{\mathtt{S}}{\to}\mathtt{U}_0 \vdash \mathtt{e} : \mathtt{U}$ *and* $\overline{\mathtt{S}}{\to}\mathtt{U}_0 \vdash \overline{\mathtt{a}}\ \mathtt{OK}$ *and* $\bullet \vdash \mathtt{b}\ \mathtt{OK}$ *and* $\mathtt{U}_0 = \bigcup_{\mathtt{a} \in \overline{\mathtt{a}}, \mathtt{b}} \mathit{rettype}(\mathtt{a})$ *, then there exists* $\mathtt{V}$ *such that* $\Gamma; \bullet \vdash [\mathtt{[}\overline{\mathtt{a}}\mathtt{,b]}/\mathtt{proceed}]\mathtt{e} : \mathtt{V}$ *and* $\mathtt{V} \mathrel{<:} \mathtt{U}$*.*

Proof. By induction on $\mathtt{e}$ with case analysis on the last typing rule used. The only interesting case is T-Proceed, which we show below.

Case T-Proceed**:** We have

$$\mathtt{e} = \mathtt{proceed}(\overline{\mathtt{e}}) \qquad \Gamma; \overline{\mathtt{S}}{\to}\mathtt{U}_0 \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \qquad \overline{\mathtt{V}} \mathrel{<:} \overline{\mathtt{S}}$$

By the induction hypothesis, there exists some $\overline{\mathtt{V}}'$ such that

$$\Gamma; \bullet \vdash [\mathtt{[}\overline{\mathtt{a}}\mathtt{,b]}/\mathtt{proceed}]\overline{\mathtt{e}} : \overline{\mathtt{V}}' \qquad \overline{\mathtt{V}}' \mathrel{<:} \overline{\mathtt{V}}.$$

By S-Trans, $\overline{\mathtt{V}}' \mathrel{<:} \overline{\mathtt{S}}$, and, by T-Woven,

$$\Gamma; \bullet \vdash \mathtt{[}\overline{\mathtt{a}}\mathtt{,b]}([\mathtt{[}\overline{\mathtt{a}}\mathtt{,b]}/\mathtt{proceed}]\overline{\mathtt{e}}) : \mathtt{U}_0$$

finishing the case.

The other cases are easy.

Lemma 5. *If* $mtype^C(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}}\to\mathtt{T}$*, then* $mbody(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0$ *for some* $\overline{\mathtt{x}}$ *and* $\mathtt{e}_0$ *where the lengths of* $\overline{\mathtt{x}}$ *and* $\overline{\mathtt{T}}$ *are the same.*

Proof. By easy induction on the derivation of $mtype^C(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}}\to\mathtt{T}$.

Lemma 6. *If* $mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}}\to\mathtt{T}$*, then* $mtype^C(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}}\to\mathtt{T}$.

Proof. By induction on the derivation of $mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}}\to\mathtt{T}$ with case analysis on the last rule used to derive $mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}}\to\mathtt{T}$. The only interesting case is when MT-SuperI is used. Then, for some $\mathtt{D}$, $\overline{\mathtt{I}}$ and i, we have $\mathtt{C}$ `extends` $\mathtt{D}$ `implements` $\overline{\mathtt{I}}$ and $mtype(\mathtt{m}, \mathtt{I}_i) = \overline{\mathtt{T}}\to\mathtt{T}$ and $\mathtt{m}$ is not present in $\mathtt{C}$. Since $\mathtt{C}$ is OK, by T-Class, it must be the case that $mtype^C(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}}\to\mathtt{T}$.

Lemma 7. *If* $mbody(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0$ *and* $mtype^C(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}}\to\mathtt{T}$*, then there exist* $\mathtt{D}$ *and* $\mathtt{U}_0$ *such that* $\mathtt{C} <: \mathtt{D}$ *and* $\mathtt{this}{:}\mathtt{C}, \overline{\mathtt{x}}{:}\overline{\mathtt{T}}; \bullet \vdash \mathtt{e}_0 : \mathtt{U}_0$ *and* $\mathtt{U}_0 <: \mathtt{T}$.

Proof. By easy induction on the derivation of $mbody(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0$.

Lemma 8. *If* $mtype(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{T}}\to\mathtt{T}_0$ *and* $\mathtt{S} <: \mathtt{T}$*, then* $mtype(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{T}}\to\mathtt{T}_0$.

Proof. We first extend the definition of *mtype* by the following rule so that the second argument can be a union type:

$$\frac{mtype(\mathtt{m}, \mathtt{U}_1) = \overline{\mathtt{T}}\to\mathtt{T}_0 \qquad mtype(\mathtt{m}, \mathtt{U}_2) = \overline{\mathtt{T}}\to\mathtt{T}_0}{mtype(\mathtt{m}, \mathtt{U}_1 \cup \mathtt{U}_2) = \overline{\mathtt{T}}\to\mathtt{T}_0} \qquad \text{(MT-Union)}$$

Note that this extension is conservative: for any $\mathtt{m}$ and $\mathtt{T}$, $mtype(\mathtt{m}, \mathtt{T})$ remains the same.

Now, we prove that, if $mtype(\mathtt{m}, \mathtt{V}) = \overline{\mathtt{T}}\to\mathtt{T}_0$ and $\mathtt{U} <: \mathtt{V}$, then $mtype(\mathtt{m}, \mathtt{U}) = \overline{\mathtt{T}}\to\mathtt{T}_0$, by induction on the derivation of $\mathtt{U} <: \mathtt{V}$ with case analysis on the last rule used.

Case S-Refl**,** S-Trans**,** S-Object**:** Trivial.

Case S-Extends**:** Let $\mathtt{U}$ and $\mathtt{V}$ be $\mathtt{C}$ and $\mathtt{D}$, respectively. The case where $\mathtt{C}$ does not have the definition of $\mathtt{m}$ is easy. Otherwise, $mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}}\to\mathtt{T}_0$ follows from T-Method (in particular, $override(\mathtt{m}, \mathtt{D}, \overline{\mathtt{T}}\to\mathtt{T}_0)$).

Case S-Implements**:** Similar to the case above.

Case S-UnionR1**,** S-UnionR2**:** Trivial.

Case S-UnionL**:** By the induction hypothesis, $mtype(\mathtt{m}, \mathtt{U}_i) = \overline{\mathtt{T}}\to\mathtt{T}_0$ for $i = 1, 2$. Then, the rule MT-Union finishes the case. □

Proof (of Theorem 1). By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$ with case analysis on the last rule used.

Case R-INVK**:** We have

$$\mathtt{e} = \mathtt{new\ C().m}(\overline{\mathtt{e}}) \qquad \mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0 \qquad \mathtt{e}' = [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ C()/this}]\mathtt{e}_0$$

for some $\mathtt{C}$, $\mathtt{m}$, $\overline{\mathtt{e}}$, $\overline{\mathtt{x}}$, and $\mathtt{e}_0$. By T-INVK and T-NEW, there exists some $\mathtt{T}_0$ such that

$$\mathtt{C} <: \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{T}} \to \mathtt{T} \qquad \Gamma; \mathtt{P} \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \overline{\mathtt{U}} <: \overline{\mathtt{T}}.$$

By Lemmas 8, 6, and 7, there exist $\mathtt{D}$ and $\mathtt{U}_0$ such that

$$\mathtt{C} <: \mathtt{D} \qquad \mathtt{this} : \mathtt{D}, \overline{\mathtt{x}} : \overline{\mathtt{T}}; \bullet \vdash \mathtt{e}_0 : \mathtt{U}_0 \qquad \mathtt{U}_0 <: \mathtt{T}.$$

By Lemmas 1 and 3,

$$\Gamma; \mathtt{P} \vdash [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ C()/this}]\mathtt{e}_0 : \mathtt{U}_0{}'$$

for some $\mathtt{U}_0{}' <: \mathtt{U}_0$. Finally, S-TRANS finishes the case.

Case R-ADVICE**:** We have

$$\mathtt{e} = \mathtt{[a}_0\mathtt{,}\overline{\mathtt{a}}\mathtt{]}(\overline{\mathtt{e}}) \qquad \mathtt{a}_0 = \mathtt{T}_0(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\mathtt{\{\ return\ e}_0\mathtt{;\ \}}$$
$$\mathtt{e}' = [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{[}\overline{\mathtt{a}}\mathtt{]/proceed}]\mathtt{e}_0$$

for some $\mathtt{a}_0$, $\overline{\mathtt{a}}$, $\overline{\mathtt{S}}$, $\mathtt{T}_0$, $\overline{\mathtt{x}}$, and $\mathtt{e}_0$. By T-WOVEN and T-ADVICE, there exist $\overline{\mathtt{a}}'$, $\mathtt{b}$, $\mathtt{U}_0$, $\overline{\mathtt{V}}$, and $\mathtt{V}_0$ such that

$$\begin{array}{lll} \overline{\mathtt{a}} = \overline{\mathtt{a}}', \mathtt{b} & \mathtt{U}_0 = \bigcup_{\mathtt{a} \in \mathtt{a}_0, \overline{\mathtt{a}}} \mathit{rettype}(\mathtt{a}) & \Gamma; \mathtt{P} \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \\ \overline{\mathtt{V}} <: \overline{\mathtt{S}} & \overline{\mathtt{x}}{:}\overline{\mathtt{S}}; \overline{\mathtt{S}} \to \mathtt{U}_0 \vdash \mathtt{e}_0 : \mathtt{V}_0 & \mathtt{V}_0 <: \mathtt{T}_0 \\ \overline{\mathtt{S}} \to \mathtt{U}_0 \vdash \overline{\mathtt{a}}'\ \mathtt{OK} & \bullet \vdash \mathtt{b\ OK}. & \end{array}$$

It is easy to show that $\mathtt{U}_0{}' \stackrel{\text{def}}{=} \bigcup_{\mathtt{a} \in \overline{\mathtt{a}}} \mathit{rettype}(\mathtt{a}) <: \mathtt{U}_0$ and, then, by Lemma 2, $\overline{\mathtt{x}}{:}\overline{\mathtt{S}}; \overline{\mathtt{S}} \to \mathtt{U}_0{}' \vdash \mathtt{e}_0 : \mathtt{V}_0$. Similarly, we have $\overline{\mathtt{S}} \to \mathtt{U}_0{}' \vdash \overline{\mathtt{a}}'\ \mathtt{OK}$. Then, by Lemma 4, there exists some $\mathtt{V}_0{}'$ such that

$$\overline{\mathtt{x}}{:}\overline{\mathtt{S}}; \bullet \vdash [\mathtt{[}\overline{\mathtt{a}}\mathtt{'},\mathtt{b]/proceed}]\mathtt{e}_0 : \mathtt{V}_0{}' \qquad \mathtt{V}_0{}' <: \mathtt{V}_0.$$

Then, by Lemmas 1 and 3, there exists some $\mathtt{V}_0{}''$ such that

$$\Gamma; \mathtt{P} \vdash [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{[}\overline{\mathtt{a}}\mathtt{'},\mathtt{b]/proceed}]\mathtt{e}_0 : \mathtt{V}_0{}'' \qquad \mathtt{V}_0{}'' <: \mathtt{V}_0{}'.$$

By S-TRANS, $\mathtt{V}_0{}'' <: \mathtt{V}_0$, finishing the case.

Case RC-INVKRECV**:** We have

$$\mathtt{e} = \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}}) \qquad \mathtt{e}_0 \longrightarrow \mathtt{e}_0{}' \qquad \mathtt{e}' = \mathtt{e}_0{}'\mathtt{.m}(\overline{\mathtt{e}})$$

for some $\mathtt{e}_0$, $\overline{\mathtt{e}}$, $\mathtt{e}_0{}'$ and $\mathtt{m}$. By T-INVK,

$$\begin{array}{lll} \Gamma; \mathtt{P} \vdash \mathtt{e}_0 : \mathtt{U}_0 & \mathtt{U}_0 <: \mathtt{T}_0 & \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{T}} \to \mathtt{T} \\ \Gamma; \mathtt{P} \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} & \overline{\mathtt{U}} <: \overline{\mathtt{T}} & \mathtt{T} = \mathtt{U} \end{array}$$

for some $\mathtt{U}_0$, $\mathtt{T}_0$, $\overline{\mathtt{T}}$, $\overline{\mathtt{U}}$ and $\mathtt{T}$. By the induction hypothesis, $\Gamma; \mathtt{P} \vdash \mathtt{e}_0{}' : \mathtt{U}_0{}'$ and $\mathtt{U}_0{}' <: \mathtt{U}_0$ for some $\mathtt{U}_0{}'$. Since $\mathtt{U}_0{}' <: \mathtt{T}_0$ by S-TRANS, we have $\Gamma; \mathtt{P} \vdash \mathtt{e}_0{}'\mathtt{.m}(\overline{\mathtt{e}}) : \mathtt{T}$ by T-INVK.

The other cases are easy.

Proof (of Theorem 2). By induction on $\mathtt{e}$. We show only main cases.

Case $\mathtt{e} = \mathtt{x}$**:** Cannot happen.
Case $\mathtt{e} = \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}$**:** By T-Invk,

$$\begin{array}{lll} \bullet;\bullet \vdash \mathtt{e}_0 : \mathtt{U}_0 & \mathtt{U}_0 \mathrel{<:} \mathtt{T}_0 & \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{T}} \to \mathtt{T} \\ \bullet;\bullet \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} & \overline{\mathtt{U}} \mathrel{<:} \overline{\mathtt{T}} & \mathtt{T} = \mathtt{U} \end{array}$$

for some $\mathtt{U}_0$, $\mathtt{T}_0$, $\overline{\mathtt{T}}$, $\overline{\mathtt{U}}$, and $\mathtt{T}$. By the induction hypothesis, we have either $\mathtt{e}_0 = \mathtt{new\ C_0()}$ for some $\mathtt{C}$ or $\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'$ for some $\mathtt{e}_0{}'$. In the latter case, we have $\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \longrightarrow \mathtt{e}_0{}'\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}$. In the former case, $\mathtt{U}_0 = \mathtt{C}_0$ and, by Lemmas 6 and 5, $\mathit{mbody}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{x}}\mathtt{.e}_0$ for some $\overline{\mathtt{x}}$ and $\mathtt{e}_0$ where the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{T}}$ are the same. Then, $\mathtt{e} \longrightarrow [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ C_0()/this}]\mathtt{e}_0$, finishing the case.
Case $\mathtt{e} = \mathtt{[}\overline{\mathtt{a}}\mathtt{](}\overline{\mathtt{e}}\mathtt{)}$**:** By T-Woven, $\overline{\mathtt{a}}$ cannot be empty. So, let $\mathtt{a}_1\mathtt{,}\ \overline{\mathtt{a}}' = \overline{\mathtt{a}}$. Then, by T-Advice, $\mathtt{a}_1$ is of the form $\mathtt{T(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e_0;\ \}}$ and the lengths of $\overline{\mathtt{x}}$ and $\overline{\mathtt{e}}$ are the same. Then, $\mathtt{e} \longrightarrow [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{[}\overline{\mathtt{a}}'\mathtt{]/proceed}]]\mathtt{e}_0$.

The other cases are easy.

C.2 Proof of Theorem 3

Theorem 4. *If* $TC(\Gamma, \mathtt{P}, \mathtt{e}) = (\mathtt{e}', \mathtt{U})$*, then* $\Gamma; \mathtt{P} \vdash \mathtt{e}' : \mathtt{U}$.

Proof. By induction on $\mathtt{e}$. We show a few interesting cases below:

Case $\mathtt{e} = \mathtt{e}_{0\langle \mathtt{T}_0 \rangle}\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}$**:** We have

$$\begin{array}{llll} \mathtt{e}' = \mathtt{e}_0{}'_{\langle \mathtt{S}_0 \rangle}\mathtt{.m(}\overline{\mathtt{e}}'\mathtt{)} & TC(\Gamma, \mathtt{P}, \mathtt{e}_0) = (\mathtt{e}_0{}', \mathtt{U}_0) & TC(\Gamma, \mathtt{P}, \overline{\mathtt{e}}) = (\overline{\mathtt{e}}', \overline{\mathtt{U}}) & \\ \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{T}} \to \mathtt{T} & \mathit{mtype}(\mathtt{m}, \mathtt{S}_0) = \overline{\mathtt{T}} \to \mathtt{T} & \mathtt{T}_0 \cup \mathtt{U}_0 \mathrel{<:} \mathtt{S}_0 & \overline{\mathtt{U}} \mathrel{<:} \overline{\mathtt{T}} \end{array}$$

for some $\mathtt{U}_0$, $\overline{\mathtt{T}}$, $\mathtt{T}$, $\overline{\mathtt{U}}$, $\mathtt{e}_0{}'$, $\overline{\mathtt{e}}'$, $\mathtt{U}_0$, $\overline{\mathtt{U}}$, and $\mathtt{S}_0$. By the induction hypothesis, $\Gamma; \mathtt{P} \vdash \mathtt{e}_0{}' : \mathtt{U}_0$ and $\Gamma; \mathtt{P} \vdash \overline{\mathtt{e}}' : \overline{\mathtt{U}}$. We have $\mathtt{U}_0 \mathrel{<:} \mathtt{S}_0$ since $\mathtt{U}_0 \mathrel{<:} \mathtt{T}_0 \cup \mathtt{U}_0$, So, by T-Invk, $\Gamma; \mathtt{P} \vdash \mathtt{e}' : \mathtt{T}$, finishing the case.
Case $\mathtt{e} = \mathtt{[}\overline{\mathtt{a}}\mathtt{,b]\,(}\overline{\mathtt{d}}\mathtt{)}$**:** We have

$$\begin{array}{ll} \overline{\mathtt{a}} = \overline{\mathtt{T}}\mathtt{(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e};\ \} & \mathtt{b} = \mathtt{T(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e_0;\ \}} \\ \mathtt{e}' = \mathtt{[}\overline{\mathtt{a}}'\mathtt{,b']\,(}\overline{\mathtt{d}}'\mathtt{)} & \\ \overline{\mathtt{a}}' = \overline{\mathtt{T}}\mathtt{(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e}'\mathtt{;\ \}} & \mathtt{b}' = \mathtt{T(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e_0}'\mathtt{;\ \}} \\ \mathtt{U}_0 = \mathtt{T} \cup \mathtt{T}_1 \cup \cdots \cup \mathtt{T}_n & \\ TC(\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}}, \overline{\mathtt{S}} \to \mathtt{U}_0, \overline{\mathtt{e}}) = (\overline{\mathtt{e}}', \overline{\mathtt{V}}) & \overline{\mathtt{V}} \mathrel{<:} \overline{\mathtt{T}} \\ TC(\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}}, \bullet, \mathtt{e}_0) = (\mathtt{e}_0{}', \mathtt{V}_0) & \mathtt{V}_0 \mathrel{<:} \mathtt{T} \\ TC(\Gamma, \mathtt{P}, \overline{\mathtt{d}}) = (\overline{\mathtt{d}}', \overline{\mathtt{U}}) & \overline{\mathtt{U}} \mathrel{<:} \overline{\mathtt{S}} \end{array}$$

for some $\overline{\mathtt{T}}$, $\overline{\mathtt{S}}$, $\overline{\mathtt{x}}$, $\overline{\mathtt{e}}$, $\mathtt{T}$, $\mathtt{e}_0$, $\overline{\mathtt{a}}'$, $\mathtt{b}'$, $\overline{\mathtt{e}}'$, $\mathtt{e}_0{}'$, $\overline{\mathtt{d}}'$, $\mathtt{U}_0$, $\overline{\mathtt{V}}$, $\mathtt{V}_0$, $\overline{\mathtt{d}}'$, and $\overline{\mathtt{U}}$. By the induction hypothesis, we have

$$\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}}; \overline{\mathtt{S}} \to \mathtt{U}_0 \vdash \overline{\mathtt{e}}' : \overline{\mathtt{V}} \quad \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}}; \bullet \vdash \mathtt{e}_0{}' : \mathtt{V}_0 \quad \Gamma; \mathtt{P} \vdash \overline{\mathtt{d}}' : \overline{\mathtt{U}}'.$$

By T-Advice, we have

$$\overline{\mathtt{S}}{\to}\mathtt{U}_0 \vdash \overline{\mathtt{a}}' \ \mathtt{OK} \quad \bullet \vdash \mathtt{b}' \ \mathtt{OK}.$$

Then, by T-Woven, $\Gamma; \mathtt{P} \vdash \mathtt{e}' : \mathtt{U}_0$, finishing the case.

Theorem 5. *If* $\Gamma; \mathtt{P} \vdash \mathtt{e} : \mathtt{U}$ *and* $\mathtt{e} \hookrightarrow \mathtt{e}'$ *and* $TC(\Gamma, \mathtt{P}, \mathtt{e}') = (\mathtt{e}'', \mathtt{U}')$*, then* $\mathtt{e} \overset{trw}{\hookrightarrow} \mathtt{e}''$.

Proof. By induction on $\mathtt{e}$. The only interesting case is when $\mathtt{e}$ is a method invocation.

Case $\mathtt{e} = \mathtt{e}_{0\langle \mathtt{T}_0 \rangle}\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}$**:** We have

$\Gamma; \mathtt{P} \vdash \mathtt{e}_0 : \mathtt{U}_0$	$\mathtt{U}_0 \ \mathtt{<:} \ \mathtt{T}_0$	$mtype(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{T}}{\to}\mathtt{T}$
$\Gamma; \mathtt{P} \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}}$	$\overline{\mathtt{U}} \ \mathtt{<:} \ \overline{\mathtt{T}}$	$\mathtt{U} = \mathtt{T}$
$\mathtt{e}' = \mathtt{e}_0{}'_{\langle \mathtt{T}_0 \rangle}\mathtt{.m(}\overline{\mathtt{e}}'\mathtt{)}$	$\mathtt{e}_0 \hookrightarrow \mathtt{e}_0{}'$	$\overline{\mathtt{e}} \hookrightarrow \overline{\mathtt{e}}'$
$\mathtt{e}'' = \mathtt{e}_0{}''_{\langle \mathtt{S}_0 \rangle}\mathtt{.m(}\overline{\mathtt{e}}''\mathtt{)}$	$TC(\Gamma, \mathtt{P}, \mathtt{e}_0{}') = (\mathtt{e}_0{}'', \mathtt{U}_0{}')$	$TC(\Gamma, \mathtt{P}, \overline{\mathtt{e}}') = (\overline{\mathtt{e}}'', \overline{\mathtt{U}}')$
$\mathtt{T}_0 \cup \mathtt{U}_0{}' \ \mathtt{<:} \ \mathtt{S}_0$	$mtype(\mathtt{m}, \mathtt{S}_0) = \overline{\mathtt{T}}{\to}\mathtt{T}$	$\overline{\mathtt{U}}' \ \mathtt{<:} \ \overline{\mathtt{T}}$

for some $\mathtt{U}_0$, $\overline{\mathtt{T}}$, $\mathtt{T}$, $\overline{\mathtt{U}}$, $\mathtt{e}_0{}'$, $\overline{\mathtt{e}}'$, $\mathtt{e}_0{}''$, $\overline{\mathtt{e}}''$, $\mathtt{U}_0{}'$, $\overline{\mathtt{U}}'$, and $\mathtt{S}_0$. By the induction hypothesis, $\mathtt{e}_0 \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e}_0{}''$ and $\overline{\mathtt{e}} \overset{\mathrm{trw}}{\hookrightarrow} \overline{\mathtt{e}}''$. We have $\mathtt{T}_0 \ \mathtt{<:} \ \mathtt{S}_0$ from $\mathtt{T}_0 \ \mathtt{<:} \ \mathtt{T}_0 \cup \mathtt{U}_0{}'$. So, $\mathtt{e} \overset{\mathrm{trw}}{\hookrightarrow} \mathtt{e}''$, finishing the case.

Proof (Theorem 3). Immediate from Theorems 4 and 5.