SUPPORTING
COVARIANT RETURN TYPES
& GENERICS
IN
TYPE RELAXED WEAVING

Tomoyuki Aotani
Japan Advanced Institute of Science and Technology
Joint work w/
Hidehiko Masuhara & Manabu Toyama
University of Tokyo
Background: AspectJ, RelaxAJ and GentleAJ

- AspectJ
  + Type relaxed weaving [Masuhara10]

- RelaxAJ
  + Support for
    * Generics
    * Covariant return types
    * `proceed`
      > multiple return types
      > argument type relaxation

- GentleAJ
Background:
AspectJ, RelaxAJ and GentleAJ

+ Type relaxed weaving [Masuhara10]

+ Support for
  * Generics
  * Covariant return types
    * proceed
      › multiple return types
      › argument type relaxation
Type relaxed weaving (TRW)\cite{Masuhara10}: Difference from AspectJ’s weaving

- Suppose we have

  ```java
  class Object{...}
  class BigInt extends Object{...}
  class Int extends Object{...}
  ```

- AspectJ and RelaxAJ(=TRW) accept

  ```java
  Int around():call(Object *.*(...)){...}
  ```

- RelaxAJ *conditionally* accept but AspectJ rejects

  ```java
  Int around():call(BigInt *.*(...)){...}
  ```
Type relaxed weaving

- Bytecode-level weaving
- Typing principle for weaving advice:

**PRINCIPLE.**
The return type of adv must be consistent with the operations that use the return value from jp.

- \( \text{jp}: \text{a join point} \)
- \( \text{adv}: \text{a piece of around advice applied to } \text{jp} \)

\( T \) around(): \( p() \{ \ldots \} \)

\( T \) must be a subtype of \( T_1, T_2 \) and \( T_3 \)
Type relaxed weaving: operations that use the return value

- Invoking a method: `o.m(a)`
  - Receiver: use type is the most general type that defines `m`
  - Argument: use type is the type appear in the signature
- Returning from the method: `return v`
- Accessing a field: `o.f=v`
- Throwing an exception: `throw v`
- Accessing an array: `a[i]=v`
Example of RelaxAJ advice: Replacing BigIntStream w/ IntStream

```java
interface Stream
{
    Object get();
}

class BigIntStream implements Stream
{
    Object get()
    {
        /*return a BigInt*/
    }
}

class IntStream implements Stream
{
    Object get()
    {
        /*return an Int*/
    }
}

bs = new BigIntStream();
o = bs.get();
s = o.toString();
/* bs is no longer used*/

IntStream around():
call(BigIntStream.new())
{
    return new IntStream();
}

Check IntStream is consistent with Stream.get()  true

Accepted

invokevirtual BigIntStream.get()

invokeinterface Stream.get()
```
Type relaxed weaving [Masuhara10]

- Bytecode weaving mechanism
- Typing rule for around advice:

**PRINCIPLE.**
The return type of \( \text{adv} \) must be consistent with the operations that use the return value from \( \text{jp} \).

- \( \text{jp} \): a join point
- \( \text{adv} \): a piece of around advice applied to \( \text{jp} \)

- Formal model: based on FJ [Igarashi01] w/ union type
- Support for Java 5 features is not considered
  - Generics and covariant return types
Go forward into Java 5: what are needed?

- Support for covariant return types
  - Changing the relaxation rule for signatures of method invocations

- Support for generics
  - Inferring erased types

We are at bytecode-level!
Example of RelaxAJ advice: Replacing BigIntStream w/ IntStream

interface Stream{ Object get(); }  
class BigIntStream implements Stream{  
    Object get(){ /*return a BigInt*/ }  
}  
class IntStream implements Stream{  
    Object get(){ /*return an Int*/ }  
}

bs = new BigIntStream();  
o = bs.get();  
s = o.toString();  
/* bs is no longer used*/

IntStream around():
    call(BigIntStream.new()){
        return new IntStream();
    }

Check IntStream is consistent with Stream.get()

true

invokevirtual BigIntStream.get()  
invokeinterface Stream.get()
Go forward into Java 5:
what are needed?

- Support for covariant return types
  - Changing the relaxation rule for signatures of method invocations

- Support for generics
  - Inferring erased types
Go forward into Java 5: what are needed?

- Support for covariant return types
  - Changing the relaxation rule for signatures of method invocations
  - Checking consistency of values derived from the return value from the join point

- Support for generics
  - Inferring erased types
  - Checking consistency of values derived from the return value from the join point

Derived values:
Let $v$ and $u$ are values. $v$ is derived from $u$ if $v$ is the return value from $x.m$ where $x$ is $u$ or some derived value from $u$
Simple support for covariant return types goes wrong

```java
class Int{
    Object toString() {...}
}
class BigInt{
    Object toString() {...}
    BigInt abs() {...}
}

interface Stream{
    Object get();
}

class BigIntStream implements Stream{
    BigInt get() {...}
}

class IntStream implements Stream{
    Int get() {...}
}

bs = new BigIntStream();
o = bs.get();
s = o.abs();
```

Check IntStream is consistent with Stream.get()

```java
IntStream around():
    call(BigIntStream.new()){
        return new IntStream();
    }
```

Not defined in Object
Simple support for covariant return types goes wrong

bs = new IntStream();
o = bs.get();
s = o.abs();
/* no bs, o and s*/

```java
class Int{
    Object toString(){...}
}
class BigInt{
    Object toString(){...}
    BigInt abs(){...}
}
@interface Stream{
    Object get();
}
class BigIntStream implements Stream{
    BigInt get(){...}
}
class IntStream implements Stream{
    Int get(){...}
}

IntStream around():
call(BigIntStream.new()){
    return new IntStream();
}
```

VerifyError
Not defined in Object
Simple support for covariant return types goes wrong

Object is used as BigInt => Error!

IntStream around():
call(BigIntStream.new()){
return new IntStream();
}
Simple support for generics goes wrong

```java
class Int{
    Object toString(){...}
}

class BigInt{
    Object toString(){...}
    BigInt abs(){...}
}

class Stream<X>{
    X val;
    Stream(X v){val=v;}
    X get(){...}
}

bs=new Stream<BigInt>(...);
o=bs.get();
s=o.abs();
```

Check `Stream<Int>` is consistent with

```java
Stream<?Obj>.get()
```

```java
Stream<Int> around():
    call(Stream<BigInt>.new(*)){
        return new Stream<Int>(...);
    }
```

BigInt Stream<BigInt>.get()

```java
?Obj Stream<?Obj>.get()
```
Simple support for generics goes wrong

```java
class Int{
    Object toString(){...}
}
class BigInt{
    Object toString(){...}
    BigInt abs(){...}
}
class Stream<X>{
    X val;
    Stream(X v){val=v;}
    X get(){...}
}

bs=new Stream<Int>(...);
o=bs.get();
s=o.abs();
/* no bs, o and s */
```

Wrong code
Simple support for generics goes wrong

```java
class Int{
    Object toString(){...}
}
class BigInt{
    Object toString(){...}
    BigInt abs(){...}
}
class Stream<X>{
    X val;
    Stream(X v){val=v;}
    X get(){...}
}

bs=new Stream<Int>(...);
o=bs.get();
s=GetInt.abs();
/* no bs, o and s */

Wrong code

?Obj is used as BigInt => Error
```

```java
Stream<Int> around():
call(Stream<BigInt>.new(*)) {
    return new Stream<Int>(...);
}
```
Our solution: checking consistency of **derived** values

- Modified typing principle (TRWc): Let $adv$ be advice and $jp$ be a join point. $adv$ can be applied to $jp$ if the return type of $adv$ is consistent w/ operations
  
  - using $ret_{jp}$ the return value from $jp$
  
  - using the derived values from $ret_{jp}$

**Derived** values:
Let $v$ and $u$ are values. $v$ is **derived** from $u$ if $v$ is the return value from $x.m$ where $x$ is $u$ or some derived value from $u$
Example: checking consistency of derived values

```
bs = new BigIntStream();
o = bs.get();
s = o.abs(); /* no bs, o and s */
```

Check
* IntStream<:Stream
* Object<:BigInt

=> Successfully reject!

```java
class Int{
    Object toString() {...}
}

class BigInt{
    Object toString() {...}
    BigInt abs(){...}
}

interface Stream{
    Object get();
}

class BigIntStream implements Stream{
    BigInt get(){...}
}
class BigIntStream implements Stream{
    Int get(){...}
}
class IntStream implements Stream{
    Int get(){...}
}
class IntStream implements Stream{
    Int around():
        call(BigIntStream.new()){ return new IntStream(); }
    
class BigIntStream implements Stream{
    BigInt get(){...}
}
```
Example: checking consistency of derived values

bs = new BigIntStream();
o = bs.get();
s = o.toString();
/* no bs, o and s */

Check
* IntStream <: Stream
* Object <: Object

=> Successfully accept!
Formalization: overview

- Featherweight Java for Relaxation w/ covariant return types (FJ Rc)
  - Simple extension to Featherweight Java for Relaxation (FJR) [Masuhara10]

- Checking consistency: constraint satisfaction
  - Generate subtyping constraints for each FJRc expression
  - If a solution is found, the woven code is (hopefully) type safe – proof: future work
Featherweight Java for Relaxation w/ Covariant Return Types (FJ Rc)

- **Syntax:** same to FJR

```
CL ::= class C extends C implements I { M }
M ::= T m(T x){ return e; }
IF ::= interface I { N }
N ::= T m (T x);
e ::= x | e.m(e) | new C() | let x = e in e | (?e:e)
T ::= C | I
U ::= T | U U
```

- **Typing rules support covariant return types**
  - Predicate `override`(m, C, T→T0)
  - Class typing rule
**Constraint generation: overview**

- **Constraint generation algorithm**
  \[ c : : (G, e) \rightarrow (P, U) \]
  
  - Typing environment \[ G : : = x:T, G \mid . \]
  
  - Expression \[ e \]
  
  - Subtyping constraint \[ P=\{\overline{p}\} \]
    where
    \[ p : : = S <: S \mid \text{retT} (m, S, S) \]
    \[ S : : = C \mid I \mid X \]
  
  - Type \[ U : : = S \mid U \cup U \cup U \]

- **Solution to a subtyping constraint** \[ P: \]
  substitution \[ [\overline{S}/\overline{X}] \] s.t. forall \[ p \in P. \] \[ [\overline{S}/\overline{X}] p \]
  
  - The return type of \( S_1.m \) is a subtype of \( S_2 \)
  
  - Variable
Constraint generation: interesting case

- Method invocation e.m(e)

\[ c(G, e_0 \cdot m(e_1, \ldots, e_n)) = \]

\[ \text{let } (P_0, U_0) = c(G, e_0) \text{ in} \]

\[ \text{let } (P, U) = c(G, \overline{e}) \text{ in} \]

\[ \text{let } T \rightarrow T = \text{mtype}(m, \text{typeOf}(e_0)) \text{ in} \]

\[ \text{let } V = \bigcup \text{ndeftypes}(m, \text{typeOf}(e_0)) \text{ in} \]

\[ (P_0 \cup P \cup \{ \overline{U} <: T \}) \cup \{ U_0 <: X_1, X_1 <: V, \text{ret}T \leq (m, X_1, X_2) \}, X_2) \]
Example: contradictions found on type-unsafe code

```javascript
Object m(){
    let s =
        (?new BigIntStrm()
            :x)
    in let i = s.get()
    in let iabs = i.abs()
    in new Object();
}
```
Example: contradictions found on type-unsafe code

```javascript
Object m() {
    let s =
        (?new BigIntStrm()
            :x)
    in let i = s.get()
    in let iabs = i.abs()
    in new Object();
}
```

```plaintext
c(x: IntStream, BigIntStrm) =
    (x: IntStream, BigIntStrm)
c(x: IntStream, IntStrm) =
    (x: IntStream, IntStrm)
c(x: IntStream, (?BigIntStrm:x)) =
    ({}) ∪ {
        BigIntStrm ∪ IntStrm
    }
```
Example: contradictions found on type-unsafe code

```javascript
Object m(){
    let s = (?new BigIntStrm()
        :x)
    in let i = s.get()
    in let iabs = i.abs()
    in new Object();
}
```

```javascript
let s = (?new BigIntStrm()
    :x)
```

```javascript
in let i = s.get()
```

```javascript
in let iabs = i.abs()
```

```javascript
in new Object();
```

```javascript
\{BigIntStrm ∪ IntStrm <: X₁₁, X₁₁ ∪ <: Strm, retT <: (get, X₁, X₂)} \}
```

```javascript
\{{}\}
```

```javascript
\{{}\}
```

```javascript
\{BigIntStrm ∪ IntStrm, s: BigIntStrm ∪ IntStrm), s} =
\{\}, BigIntStrm ∪ IntStrm)
```

```javascript
mtype (get, typeOf (s)) = () ⇒ BigInt
```

```javascript
∪mdeftypes (get, BigIntStrm) = Strm
```

```javascript
\c ((x: IntStrm, s: BigIntStrm ∪ IntStrm), s.get () =
\{ BigIntStrm ∪ IntStrm <: X₁₁, X₁₁ ∪ <: Strm, retT <: (get, X₁, X₂) \}, X₂)
```
Example: contradictions found on type-unsafe code

Object m(){
    return
    let s =
        (?new BigIntStrm()
            :x)
    in let i = s.get()
    in let iabs = i.abs()
    in new Object();
}

c((x:IntStrm,s:BigIntStrm∪IntStrm,i:X_2),i)=
    ({{},X_2})

mtype(abs,typeOf(s))=()→BigInt

Undeftypes(abs,BigInt)=BigInt
c((x:IntStrm,s:BigIntStrm∪IntStrm,i:X_2),i.abs())=
    ({{X_2<<X_3,X_3<<BigInt,retT<:(abs,X_3,X_4)}},X_4)}
Example: contradictions found on type-unsafe code

```
Object m(){
  return {
    let s = (?new BigIntStrm() :x)
    in let i = s.get()
    in let iabs = i.abs()
    in new Object();
  }
}
```

```
BigIntStrm ∪ IntStrm :< X₁,
X₁ :< Strm, retT (get, X₁, X₂)
{X₂ :< X₃, X₃ :< BigInt, retT (abs, X₃, X₄)}
```

```
BigIntStrm ∪ IntStrm :< X₁ :< Strm => X₁ = Strm
retT (get, X₁, X₂) = retT (get, Strm, X₂) => X₂ = Object
X₂ :< X₃ :< BigInt = Object :< X₃ :< BigInt => False
```
Conclusions and future work

- Type relaxed weaving w/ covariant return types (and generics)
  - Checking derived values is necessary
- Constraint generation algorithm for FJ Rc
  - Changes from FJR: just about return types
- Future work
  - Proving type safety of FJRc and soundness of the algorithm
  - Implementation
Example: checking consistency of derived values

```java
class Int{
    Object toString(){...}
}
class BigInt{
    Object toString(){...}
    BigInt abs(){...}
}
interface Stream{
    Object get();
}
class BigIntStream implements Stream{
    BigInt get(){...}
}
class IntStream implements Stream{
    Int get(){...}
}

bs = new BigIntStream();
o = bs.get();
s = o.abs(); /* no bs, o and s */

Check
* IntStream <: BigIntStream
* BigInt <: BigInt
=> Successfully reject!

IntStream around():
    call(BigIntStream.new()){ return new IntStream();

/* no bs, o and s */
```
Example: contradictions found on type-safe code

Object m(){
    return
    let s =
        (?new BigIntStrm()
            :x)
    in let i = s.get()
    in let t = i.toStr()
    in new Object();
}

BigIntStrm ∪ IntStrm <: X₁, X₁ <: Strm, retT <<< get, X₁, X₂ >>>

{X₂ <: X₃, X₃ <: Object, retT <<< toStr, X₃, X₄ >>>}

retT <<< get, X₁, X₂ >>> = retT <<< get, Strm, X₂ >>> => X₂ = Object
retT <<< toStr, X₃, X₄ >>> = retT <<< toStr, Object, X₄ >>> => X₄ = Strm

X₁ = Strm
X₂ = Object
X₃ = Object
X₄ = Strm