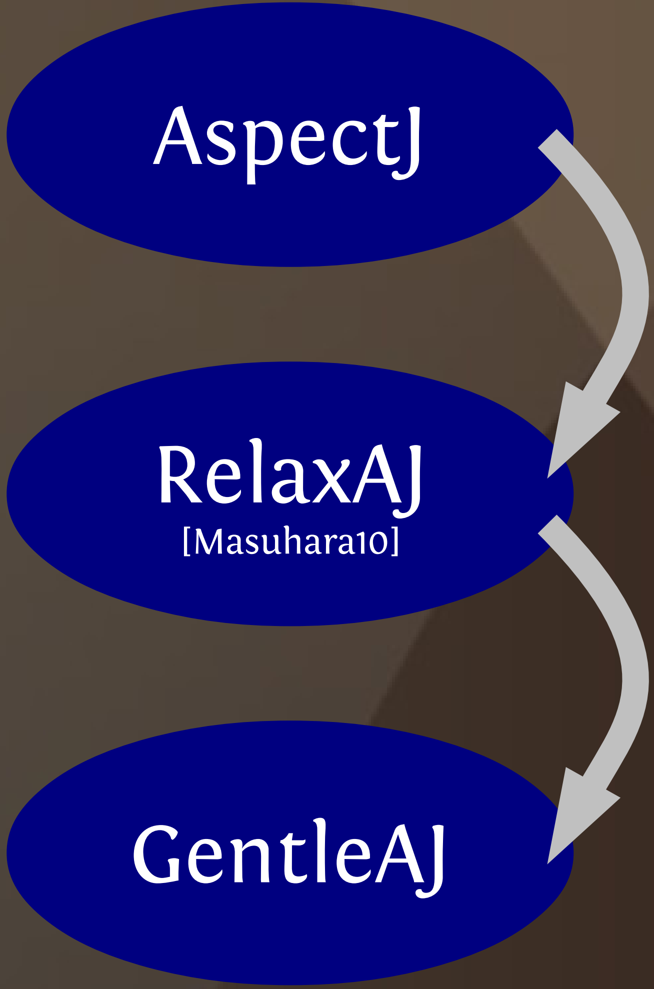
Supporting Covariant Return Types & Generics in Type Relaxed Weaving

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

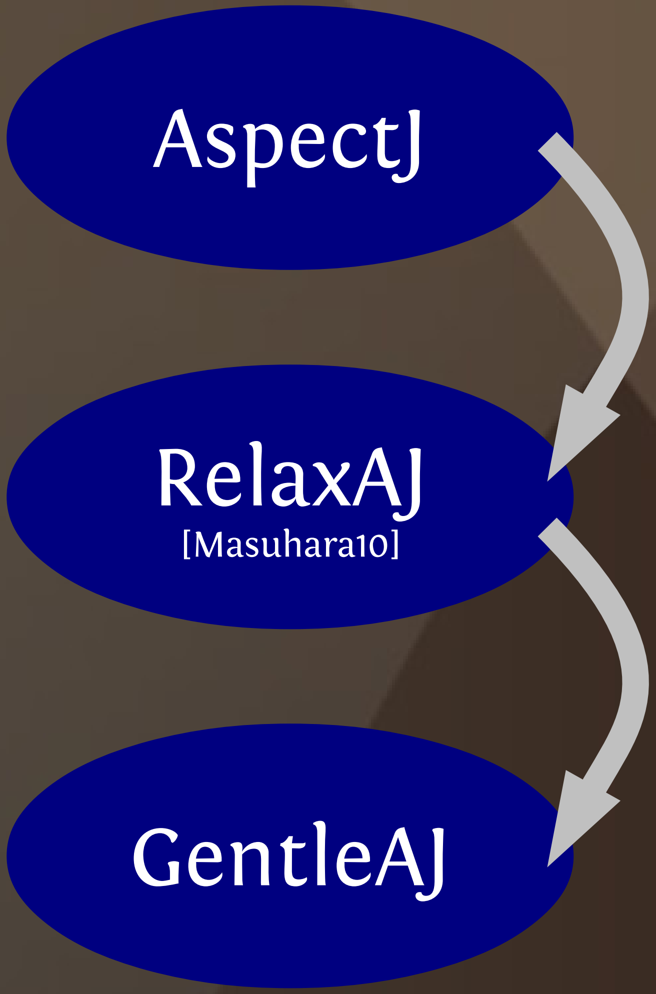
RelaxAJ
[Masuhara10]

GentleAJ

+ Type relaxed weaving[Masuhara10]

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

Background:
AspectJ, RelaxAJ and GentleAJ



AspectJ

RelaxAJ
[Masuhara10]

GentleAJ

+ Type relaxed weaving[Masuhara10]

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

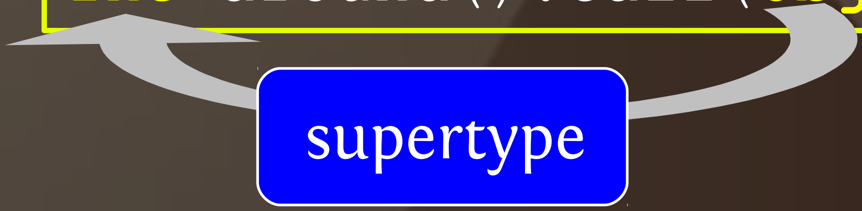
Type relaxed weaving (TRW)[Masuhara10]*: Difference from AspectJ’s weaving*

- Suppose we have

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

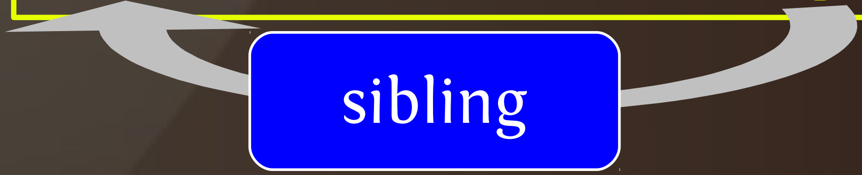
- AspectJ and RelaxAJ(=TRW) accept

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



- RelaxAJ *conditionally* accept but AspectJ rejects

```
Int around():call(BigInt *.*(...)){...}
```

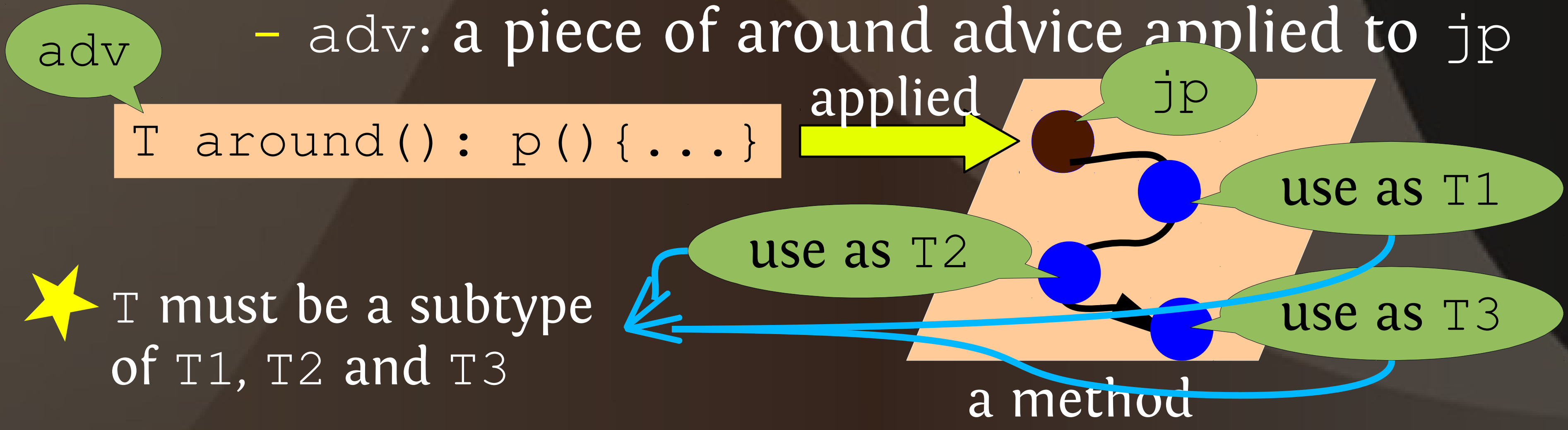


Type relaxed weaving [Masuhara10]

- 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`.

  - `jp`: a join point
  - `adv`: a piece of around advice applied to `jp`



```
T around(): p(){...}
```

★ `T` must be a subtype of `T1`, `T2` and `T3`

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
- Returnning 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

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

Sibling of `BigIntStream`

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

Check `IntStream` is consistent with `Stream.get()` } true

Accepted

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

`invokevirtual BigIntStream.get()`

`invokeinterface Stream.get()`

Type relaxed weaving[Masuhara10]

- Bytecode weaving mechanism
- Typing rule for around advice:

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

  - `jp`: a join point
  - `adv`: a piece of around advice applied to `jp`
- Formal model: based on FJ[Igarashi01] w/ union type
- Support for Java 5 features is not considered
  - Generics and covariant return types

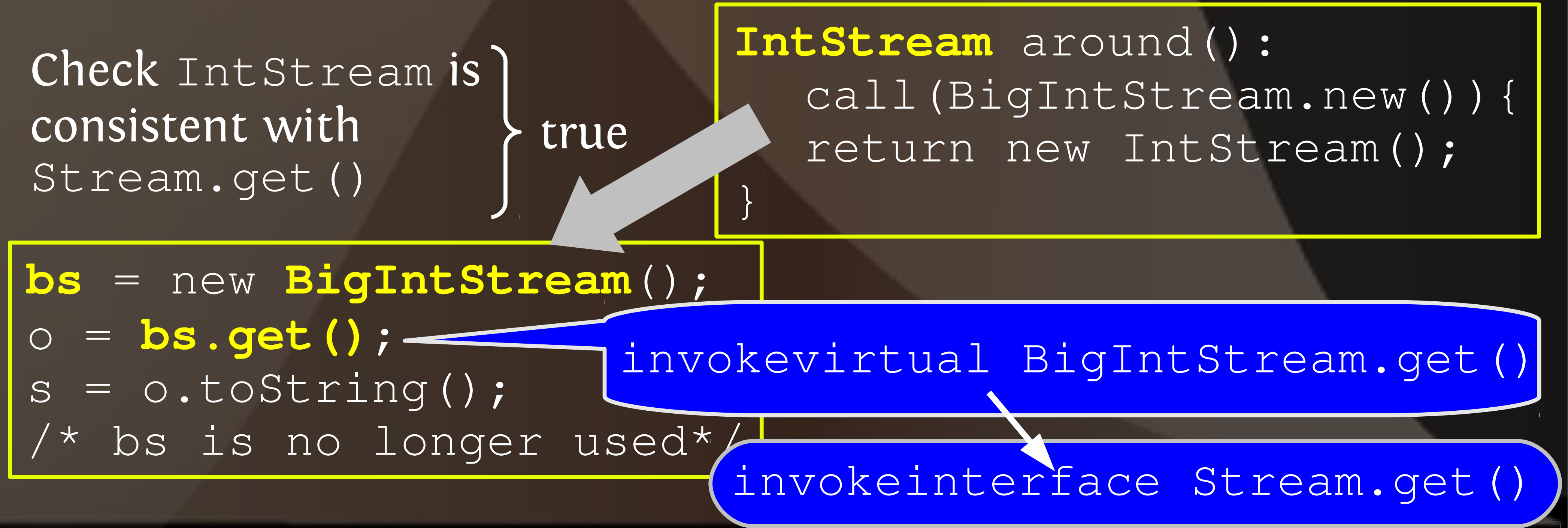
Go forward into Java 5: what are needed?

- 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*/ }
}
```



Check `IntStream` is consistent with `Stream.get()` } true

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

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

```
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

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

BigInt BigIntStream.get()

Object Stream.get()

Check `IntStream` is consistent with `Stream.get()`

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

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

Not defined in Object

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

Simple support for co return types wrong

VerifyError

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

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

```
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(){...}
}
```

Not defined in Object

Simple support for covariant return types is wrong

VerifyError

```
bs = new IntStream();
o = bs.get();
s = o.abs();
/*
```

Object Stream.get()

BigInt BigInt.abs()

Object is used as BigInt
=> Error!

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

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

Not defined in Object

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

Simple support for generics goes wrong

```
?Obj=? extends Object
```

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

```
BigInt Stream<BigInt>.get()
```

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

Check `Stream<Int>` is consistent with `Stream<?Obj>.get()`

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

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

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

Simple support for generics goes wrong

```
?Obj=? extends Object
```

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

Wrong code

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


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

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

Simple support for generics goes wrong

?Obj=? extends Object

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

?Obj Stream<?Obj>.get()

BigInt BigInt.abs()

Wrong code

?Obj is used as BigInt
=> Error

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

```
               ream<X>{

   Stream(X v){val=v;}
   X get(){...}
}
```

```
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 $\mathtt{ret}_{\mathtt{jp}}$
  - using the derived values from $\mathtt{ret}_{\mathtt{jp}}$

the return value from `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!

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

Object Stream.get()

BigInt BigIntStream.get()

BigInt BigInt.abs()

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

                 ream{
   Object get();
}
class BigIntStream
   implements Stream{
  BigInt get(){...}
}
class IntStream
   implements Stream{
  Int 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!

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

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

Object Stream.get()

BigInt BigIntStream.get()

Objet BigInt.toString()

Object Object.toString()

Formalization: overview

- Featherweight Java for Relaxation w/ covariant return types (FJRc)
  - 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 (FJRc)

- Syntax: same to FJR

$$\begin{array}{lll}
\texttt{CL} & \texttt{::=} & \texttt{class C extends C implements }\overline{\texttt{I}}\texttt{ \{ }\overline{\texttt{M}}\texttt{ \}}\\
\texttt{M} & \texttt{::=} & \texttt{T m(}\overline{\texttt{T}}\texttt{ }\overline{\texttt{x}}\texttt{)\{ return e; \}}\\
\texttt{IF} & \texttt{::=} & \texttt{interface I \{ }\overline{\texttt{N}}\texttt{ \}}\\
\texttt{N} & \texttt{::=} & \texttt{T m (}\overline{\texttt{T}}\texttt{ }\overline{\texttt{x}}\texttt{);}\\
\texttt{e} & \texttt{::=} & \texttt{x | e.m(}\overline{\texttt{e}}\texttt{) | new C() | let x = e in e | (?e:e)}\\
\texttt{T} & \texttt{::=} & \texttt{C | I}\\
\texttt{U} & \texttt{::=} & \texttt{T | U } \cup \texttt{ U}
\end{array}$$

non-deterministic choice

woven advice

- Typing rules support covariant return types
  - Predicate $\texttt{override(m,C,}\overline{\texttt{T}}\rightarrow\texttt{T0)}$
  - Class typing rule

Constraint generation: overview

- Constraint generation algorithm
  `c ::(G,e)→(P,U)`
  - Typing environment `G ::= x:T,G | .`
  - Expression `e`
  - Subtyping constraint `P={p̄}`
    where
    `p ::= S <: S |` **`retT`**$_{\texttt{<:}}$ **`(m,S,S)`**
    `S ::= C | I | X`
  - Type `U ::= S | U ∪ U`
- Solution to a subtyping constraint `P`:
  substitution $[\overline{S}/\overline{X}]$ s.t. forall `p∈P.` $[\overline{S}/\overline{X}]$`p`

the return type of S1.m
is a subtype of S2

variable

Constraint generation: interesting case

- Method invocation e.m($\overline{\text{e}}$)

$$
\begin{aligned}
&\texttt{c}(\texttt{G},\texttt{e}_0\texttt{.m}(\texttt{e}_1,\ldots,\texttt{e}_n)) = \\
&\quad \texttt{let }(\texttt{P}_0,\texttt{U}_0) = \texttt{c}(\texttt{G},\texttt{e}_0)\texttt{ in} \\
&\quad \texttt{let }(\overline{\texttt{P}},\overline{\texttt{U}}) = \texttt{c}(\texttt{G},\overline{\texttt{e}})\texttt{ in} \\
&\quad \texttt{let }\overline{\texttt{T}} \to \texttt{T} = \texttt{mtype}(\texttt{m},\texttt{typeOf}(\texttt{e}_0))\texttt{ in} \\
&\quad \texttt{let V} = \bigcup \texttt{mdeftypes}(\texttt{m},\texttt{typeOf}(\texttt{e}_0))\texttt{ in} \\
&\quad (\texttt{P}_0 \cup \overline{\texttt{P}} \cup \{\overline{\texttt{U}} <: \overline{\texttt{T}}\} \cup \{\underbrace{\texttt{U}_0 <: \texttt{X}_1, \texttt{X}_1 <: \texttt{V}}_{\text{receiver's type can be relaxed}}, \underbrace{\texttt{retT}_{<:}(\texttt{m},\texttt{X}_1,\texttt{X}_2)}_{\text{checking derived values}}\} \\
&\quad ,\texttt{X}_2)
\end{aligned}
$$

non-relaxed type of $\texttt{e}_0$

least upper bound of the types that define `m`

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();
}
```

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: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

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

{}

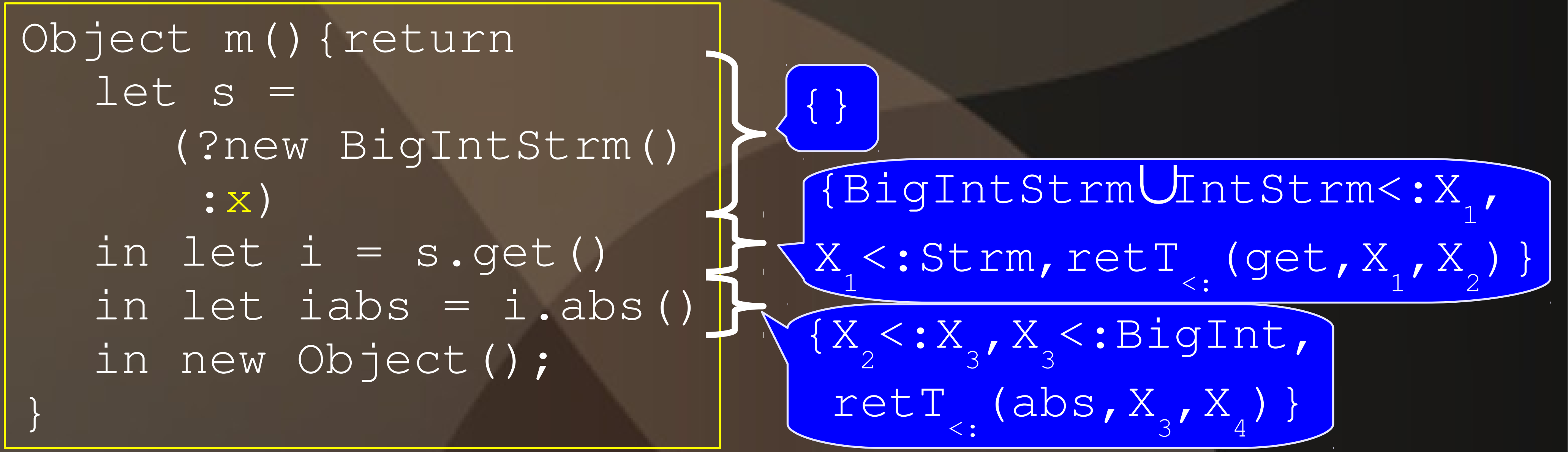
{BigIntStrm∪IntStrm<:X_1, X_1<:Strm, retT$_{<:}$(get,X_1,X_2)}

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

mtype(get,typeOf(s))=()→BigInt
∪mdeftypes(get,BigIntStrm)=Strm

c((x:IntStrm,s:BigIntStrm∪IntStrm),s.get())=
({BigIntStrm∪IntStrm<:X_1,X_1<:Strm,retT$_{<:}$(get,X_1,X_2}
,X_2)

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();
}
```

$\{\}$

$\{BigIntStrm \cup IntStrm <: X_1, X_1 <: Strm, retT_{<:}(get, X_1, X_2)\}$

$\{X_2 <: X_3, X_3 <: BigInt, retT_{<:}(abs, X_3, X_4)\}$

$c((x{:}IntStrm, s{:}BigIntStrm \cup IntStrm, i{:}X_2), i) =$

$\quad (\{\}, X_2)$

$mtype(abs, typeOf(s)) = () \to BigInt$

$\cup mdeftypes(abs, BigInt) = BigInt$

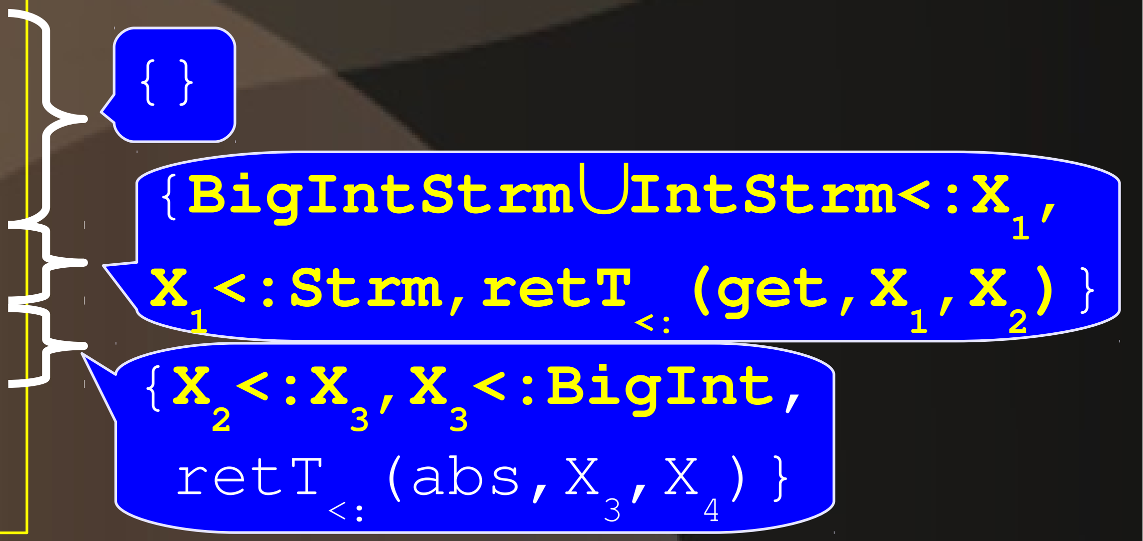
$c((x{:}IntStrm, s{:}BigIntStrm \cup IntStrm, i{:}X_2), i.abs()) =$

$\quad (\{X_2 <: X_3, X_3 <: BigInt, retT_{<:}(abs, X_3, X_4)\}$

$\quad , X_4)$

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();
}
```



{ }

$\{\mathtt{BigIntStrm} \cup \mathtt{IntStrm} <: X_1, X_1 <: \mathtt{Strm}, \mathtt{retT}_{<:}(\mathtt{get}, X_1, X_2)\}$

$\{X_2 <: X_3, X_3 <: \mathtt{BigInt}, \mathtt{retT}_{<:}(\mathtt{abs}, X_3, X_4)\}$

$\mathtt{BigIntStrm} \cup \mathtt{IntStrm} <: X_1 <: \mathtt{Strm} \Rightarrow X_1 = \mathtt{Strm}$

$\mathtt{retT}_{<:}(\mathtt{get}, X_1, X_2) = \mathtt{retT}_{<:}(\mathtt{get}, \mathtt{Strm}, X_2) \Rightarrow X_2 = \mathtt{Object}$

$X_2 <: X_3 <: \mathtt{BigInt} = \mathtt{Object} <: X_3 <: \mathtt{BigInt} \Rightarrow$ **False**

Conclusions and future work

- Type relaxed weaving w/ covariant return types (and generics)
  - Checking derived values is necessary
- Constraint generation algorithm for FJRc
  - 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

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

BigInt BigIntStream.get()

BigInt BigInt.abs()

Check
- IntStream<:
      BigIntStream
- BigInt<:BigInt

=> Successfully reject!

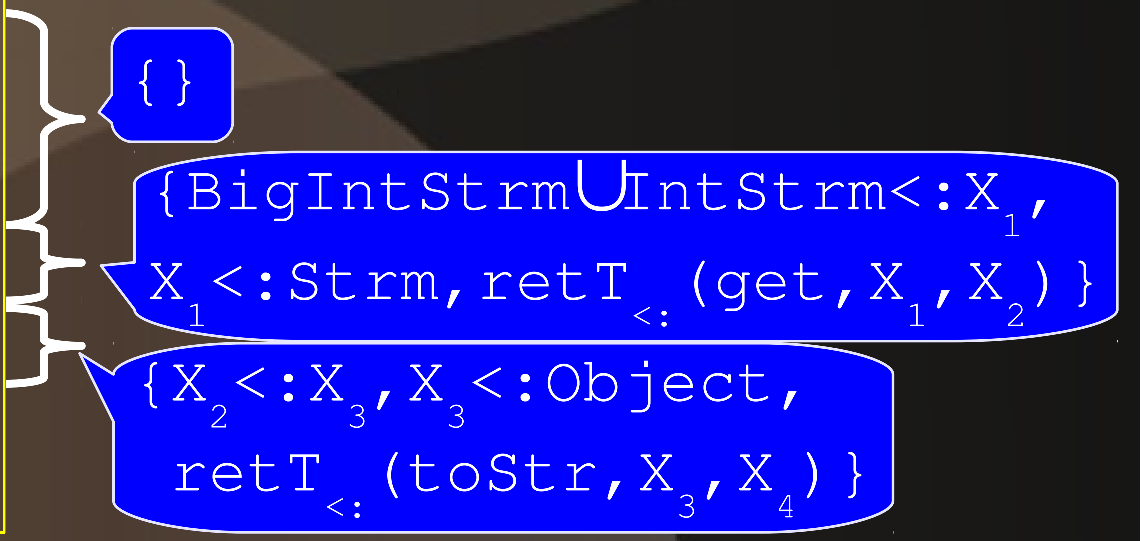
```
IntStream around():
   call(BigIntStream.new()){
   return new IntStream();
}
```

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

                 ream{
    Object get();
}
class BigIntStream
    implements Stream{
   BigInt get(){...}
}
class IntStream
    implements Stream{
   Int get(){...}
}
```

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();
}
```



$\{\}$

$\{\mathrm{BigIntStrm} \cup \mathrm{IntStrm} <: X_1, X_1 <: \mathrm{Strm}, \mathrm{retT}_{<:}(\mathrm{get}, X_1, X_2)\}$

$\{X_2 <: X_3, X_3 <: \mathrm{Object}, \mathrm{retT}_{<:}(\mathrm{toStr}, X_3, X_4)\}$

$\mathrm{BigIntStrm} \cup \mathrm{IntStrm} <: X_1 <: \mathrm{Strm} \Rightarrow X_1 = \mathrm{Strm}$

$\mathrm{retT}_{<:}(\mathrm{get}, X_1, X_2) = \mathrm{retT}_{<:}(\mathrm{get}, \mathrm{Strm}, X_2) \Rightarrow X_2 = \mathrm{Object}$

$X_2 <: X_3 <: \mathrm{Object} = \mathrm{Object} <: X_3 <: \mathrm{Object} \Rightarrow X_3 = \mathrm{Object}$

$\mathrm{retT}_{<:}(\mathrm{toStr}, X_3, X_4) = \mathrm{retT}_{<:}(\mathrm{toStr}, \mathrm{Object}, X_4) \Rightarrow X_4 = \mathrm{Str}$