An Approach for Advice Composition by a Composable Construct

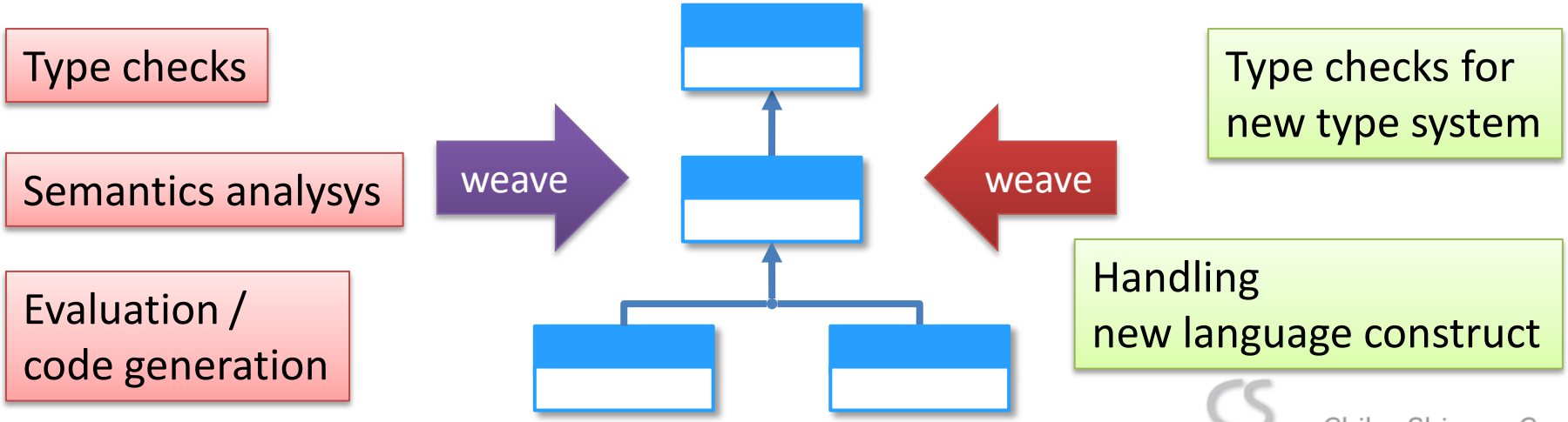
Fuminobu Takeyama and Shigeru Chiba
Tokyo Institute of Technology

AOAsia/Pacific '10 at University of Tokyo

Chiba Shigeru Group
Tokyo Tech

An aspect oriented development scenario 1/3

- A compiler is one of the best case study of AOP
  - JastAdd [T. Ekman, et al, OOPSLA 07]
    - Classes represents ASTs and aspects implement features
  - Programmers can extend compiler by implementing additional aspects
    - to support their own language extension



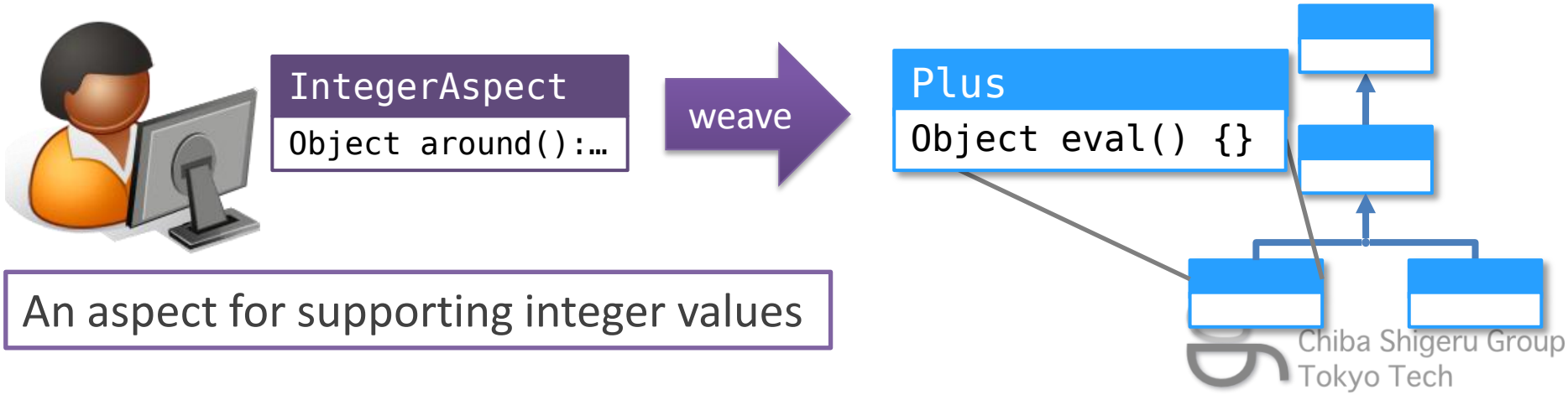
Chiba Shigeru Group
Tokyo Tech

An aspect oriented development scenario 2/3

- Alice extends the interpreter by aspects
  - Addition of integer values are implemented by IntegerAspect
- Override `Plus.eval()` method by an advice

```
new Plus(new Constant(1), new Constant(2)).eval()
```

  - → 3



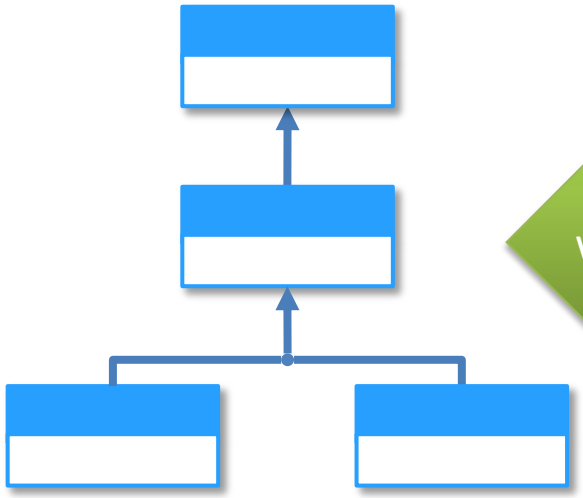
An aspect for supporting integer values

An aspect oriented development scenario 3/3

- Bob extends the original interpreter by StringAspect
  - Concatination of character strings using + operator

```
new Plus(new Constant("Hello "), new Constant("world!")).eval()
```

  - → “Hello world!"



weave

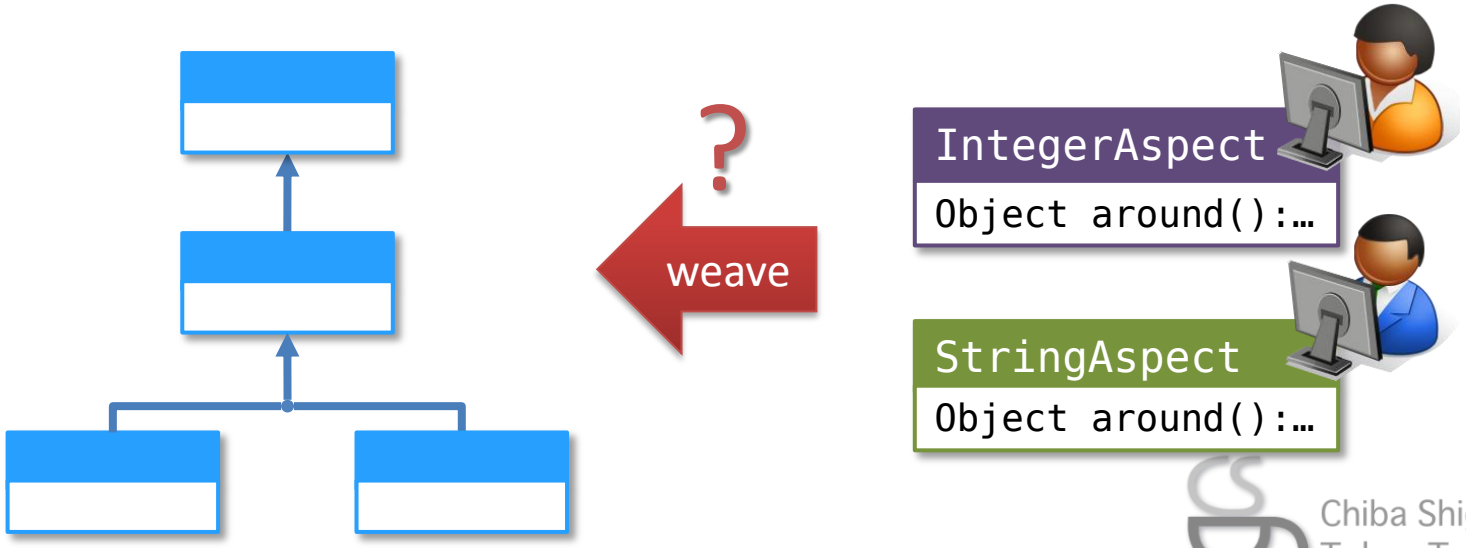
```
StringAspect
Object around():…
```

An aspect for supporting character strings

Chiba Shigeru Group
Tokyo Tech

Question 1

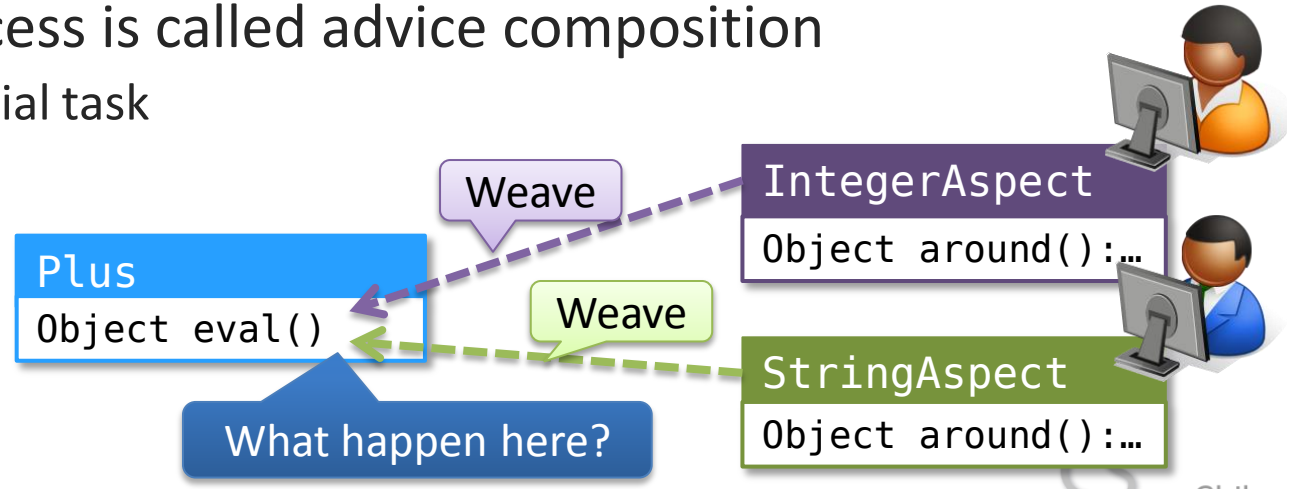
- How can we get an interpreter supporting both integers and character strings?
- An ideal approach of AOP
  - just by compiling Alice's and Bob's aspects together



Chiba Shigeru Group
Tokyo Tech

No satisfactory solution in AspectJ

- Those advices conflict at a join point
  - Conflict: multiple advices are woven into the same joinpoint
- We need more powerful composition mechanism
  - than declare precedence
  - to obtain expected behavior at the join point
  - This process is called advice composition
    - Non-trivial task



Chiba Shigeru Group
Tokyo Tech

A naive and incomplete solution 1/3

- An advice below works well just by compiled together
  - implemented as composable as possible in AspectJ

StringAspect

```
Object around(Plus t):
   target(t) && execution(Object Plus.eval()) {

  if ( At least one of operands is String ) {

    return Converts and concatenates both operands ;

  } else {

    return Invoke another advice and pass responsibility to it ;
}}
```

I can handle this type

A naive and incomplete solution 2/3

- An advice must be aware of composition with others
  - maybe unknown yet

StringAspect

```
Object around(Plus t):
   target(t) && execution(Object Plus.eval()) {

  if ( At least one of operands is String ) {

    return Converts and concatenates both operands ;

  } else {

    return Invoke another advice and pass responsibility to it ;
}}
```

A Protocol for working together

A operands may be an integer

A Protocol for working together

A naive and incomplete solution 3/3

- Composition is another crosscutting concern
  - scatters over aspects

IntegerAspect

```
aspect IntegerAspect {
  Object around(Plus t): target(t) && execution(Object Plus.eval()) {
    Object left = t.getLeft().eval();
    Object right = t.getLeft().eval();
    if (left instanceof Integer && right instanceof Integer) {
      return (Integer)left + (Integer)right;
    } else {
      return proceed(t);
}}}
```

composition code!

StringAspect

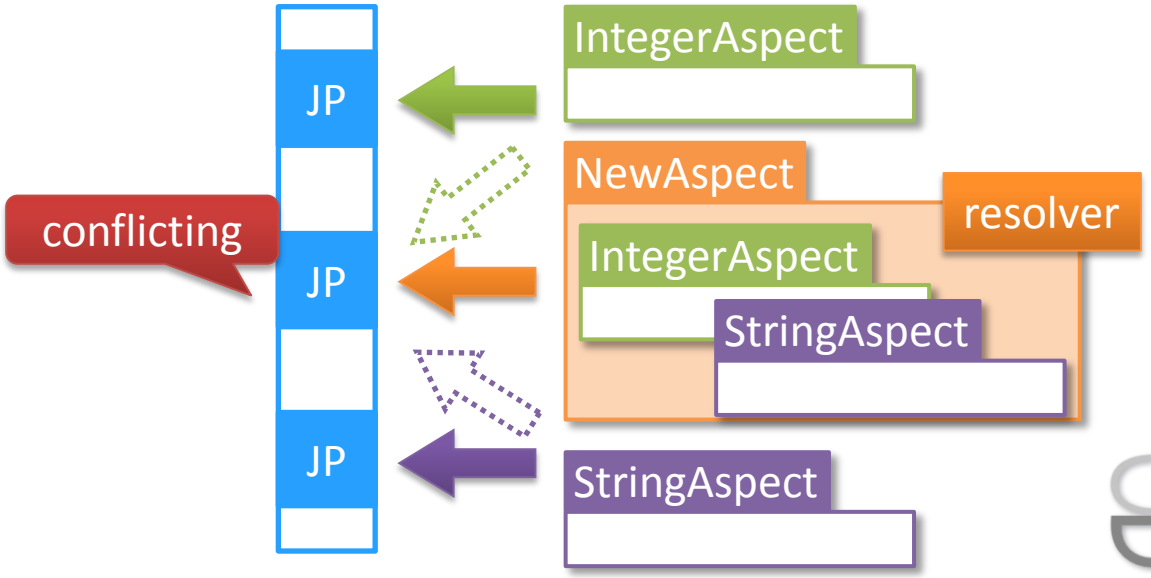
```
aspect StringAspect {
  Object around(Plus t): target(t) && execution(Object Plus.eval()) {
    Object left = t.getLeft().eval();
    Object right = t.getLeft().eval();
    if (left instanceof String || right instanceof String) {
      return left.toString() + right.toString();
    } else {
      return proceed(t);
}}}
```

composition code!

Chiba Shigeru Group
Tokyo Tech

Airia: an extension of AspectJ

- Describe composed behaviour by a resolver
  - A resolver is new kind of advice
  - Manually implemented by programmers who reuse advices
- A resolver is executed only at join points when given advices conflict



Chiba Shigeru Group
Tokyo Tech

1: Each advice is unaware of composition

- Append an advice name to each advice

IntegerAspect

Advice name

```
aspect IntegerAspect {
   Object plusEvalInt around(Plus t):
        target(t) && execution(Object Plus.eval()) {

      return (Integer)t.getLeft().eval() + (Integer)t.getRight().eval();
   }
}
```

No composition code

StringAspect

```
aspect StringAspect {
   Object plusEvalStr around(Plus t):
        target(t) && execution(Object Plus.eval()) {

      return (String)t.getLeft().eval() + (String)t.getRight().eval();
   }
}
```

No composition code

2: Implementing composition by a resolver

IntegerStringAspect

Conflicting advice

```
Object resolver plusEvalIntStr(Plus t)
        and(IntegerAspect.plusEvalInt(t), StringAspect.plusEvalStr) {

  if ( Both operands are Integer values ) {

        //Invokes only IntegerAspect

    return [IntegerAspect.plusEvalInt].proceed(t);

  } else if ( Both operands are string characters ) {

        //Invoke only StringAspect

    return [StringAspect.plusEvalStr].proceed(t);

  } else { //One of operands is string caracters

    return Concatenate both operands as string caracters ;

}}
```

Select an advice to invoke

merged behavior of the two aspect

Our constructs in detail

- Resolver
  - and/or clause specifies the join points of the resolver
    - has no pointcut

```
Object resolver name() and|or(advice1, advice2) {}
```

- *Proceed call with precedence* executes a remaining advice
  - declares precedence order among advices &
    - depending on dynamic context
  - remove unnecessary advices at that call

```
[advice1, advice2].proceed();
```

Chiba Shigeru Group
Tokyo Tech

Question 2

- Is it enough that composition is separated into a module?
  - Several research activities like this exist
- If Alice and Bob had resolvers, what happens?
  - The resolvers might conflict each other

Chiba Shigeru Group
Tokyo Tech

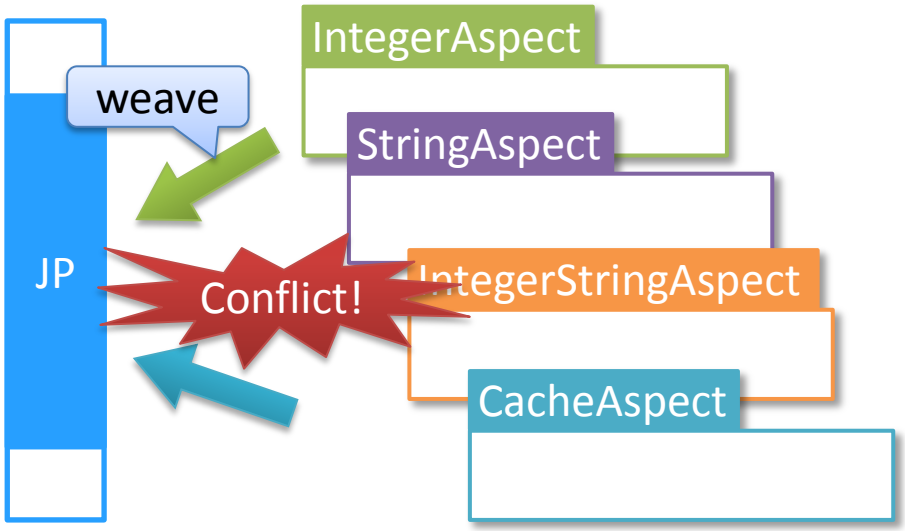
Resolvers are composable

- A resolver is a special around advice
  - Conflict among normal advices and resolvers can be resolved by resolvers
  - Resolvers can be controled by [].proceed
    - in the same way as normal advice
- Declarative precedence order
  - A proceed call declares A < (precedes) B
    Another declares A < C < B
  - Compiler can determine the total order: A, C, B

Chiba Shigeru Group
Tokyo Tech

A new advice for the interpreter

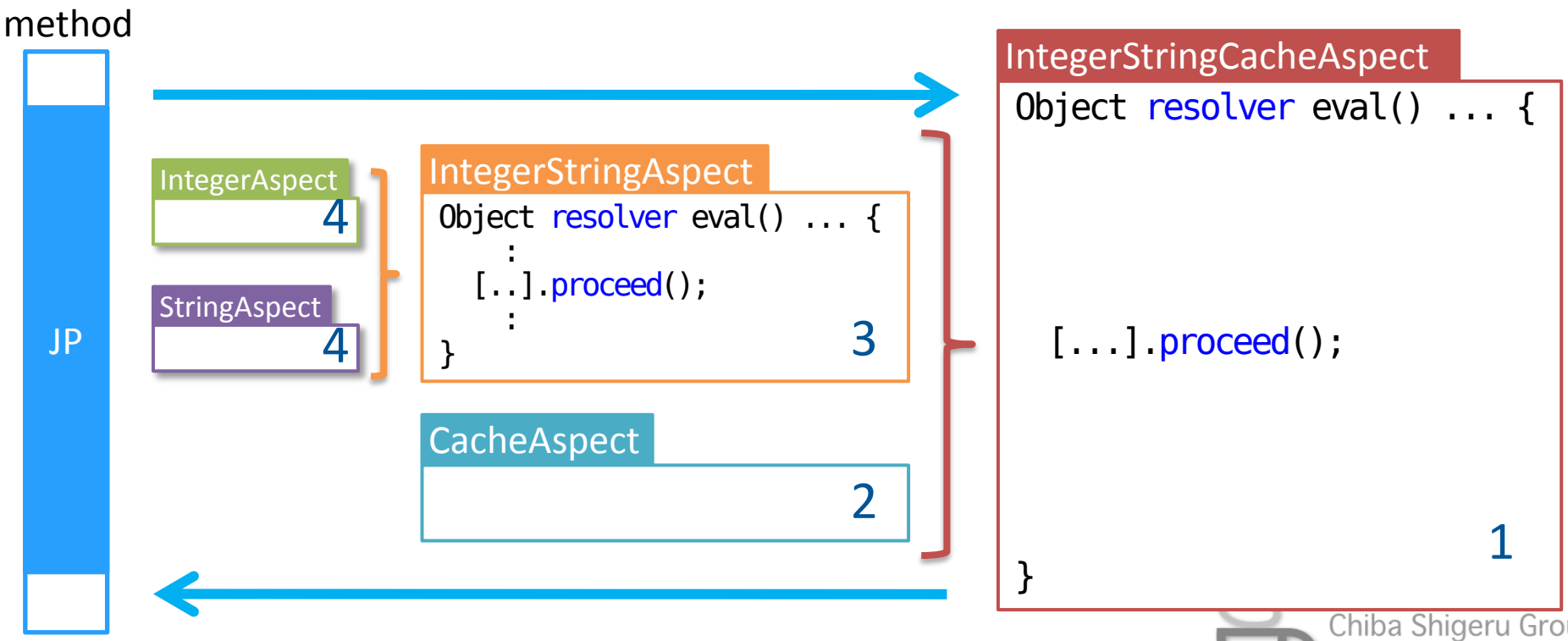
- CacheAspect saves evaluated value of expressions
  - 3 aspects and a resolver conflict at the execution of Plus.eval()



Chiba Shigeru Group
Tokyo Tech

Advices are composed in a hierarchical manner

- Implements new resolver for composition of CacheAspect and IntegerStringAspect



Chiba Shigeru Group
Tokyo Tech

Compile time check of conflict resolution

- All conflict among advices (and resolvers) must be resolved by resolvers consistently
  - Precedence order is a bit complex
  - No default precedence
- Limitation for enabling compile time check
  - Static conflict: overlap of shadow
  - Checks execition order for evary possible control path
    - Our checking algorithm is conservative

Chiba Shigeru Group
Tokyo Tech

Ideas of Airia

- Aspects are free from composition code
  - Separating composition code into a resolver
- Resolvers are composable
  - Conflict of resolvers can be resolved by other resolvers in the same way
- Precedence order is checked statically

Chiba Shigeru Group
Tokyo Tech

Related work

- Meta-programming approach
  - POPART [T. Dinkelaker, et al, AOSD 09],
    JAsCo [D. Suvée, et al, AOSD 03],
    OARTA [A. Marot, et al, AOSD 10]
  - They do not support composition among meta code and advice
- Airia is inspired by
  - Traits [N. Schärli, et al, ECOOP 03]
  - Context-Aware Composition Rules
    [A. Marot, et al, DSAL 08 and SPLAT 08]

Chiba Shigeru Group
Tokyo Tech

Conclusion

- Airia enables more powerful advice composition by a composable construct
  - Composed behavior of conflicting advices is separated into resolvers
  - Composition of resolvers and advices is possible
- The Airia compiler is available from
  - http://www.csg.is.titech.ac.jp/projects/airia/
- For more detail,
  please refer to our paper published
  in Software Composition 2010