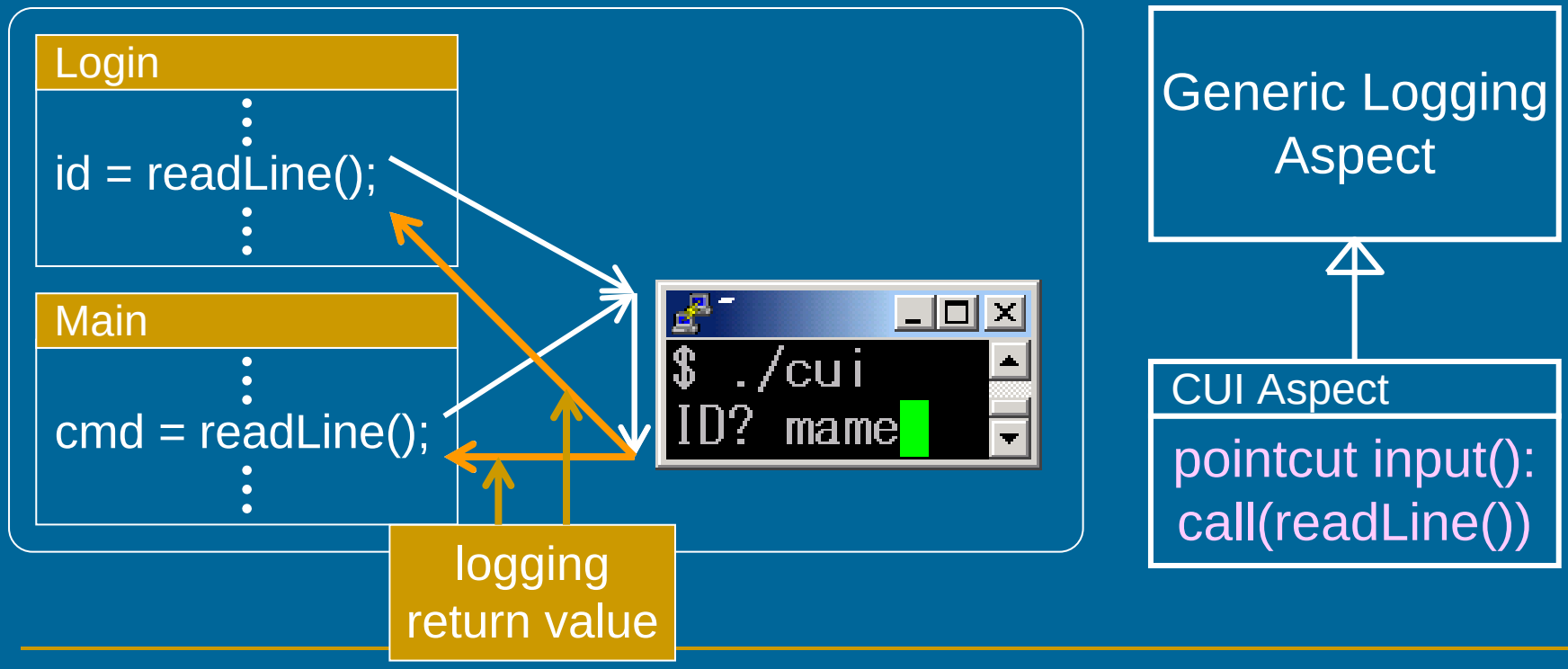
Continuation Join Points

Yusuke Endoh, Hidehiko Masuhara, Akinori Yonezawa
(University of Tokyo)

Background: Aspects are reusable in AspectJ (1)

- Example: A generic logging aspect
  - can log user inputs in a CUI program
  - by defining a pointcut



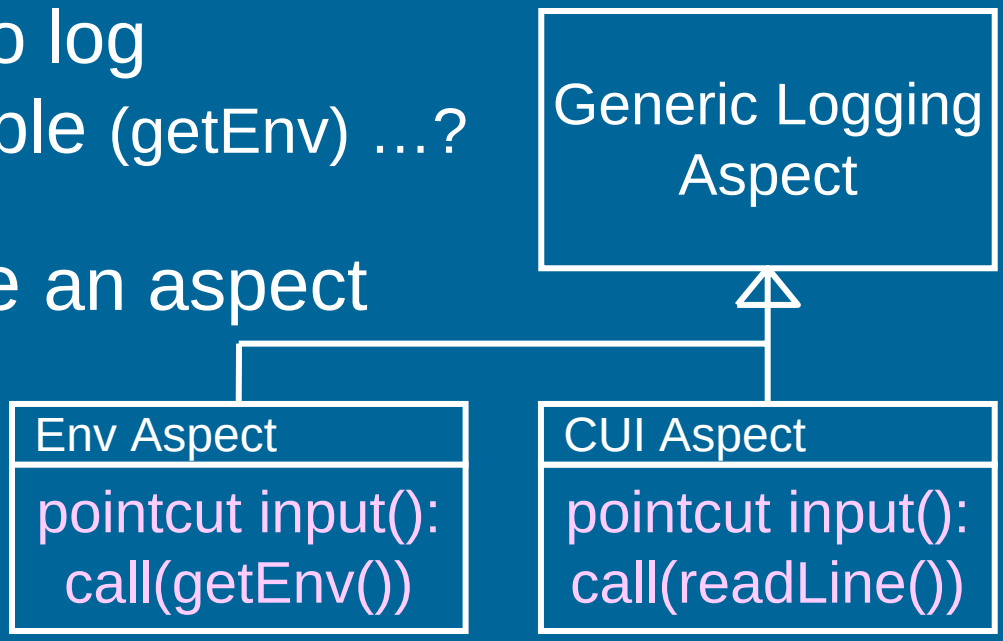
Background: Aspects are reusable in AspectJ (2)

- Example: A generic logging aspect
  - **can also log environment variable**
  - by also defining a pointcut

Q. Now, if we want to log environment variable (getEnv) …?

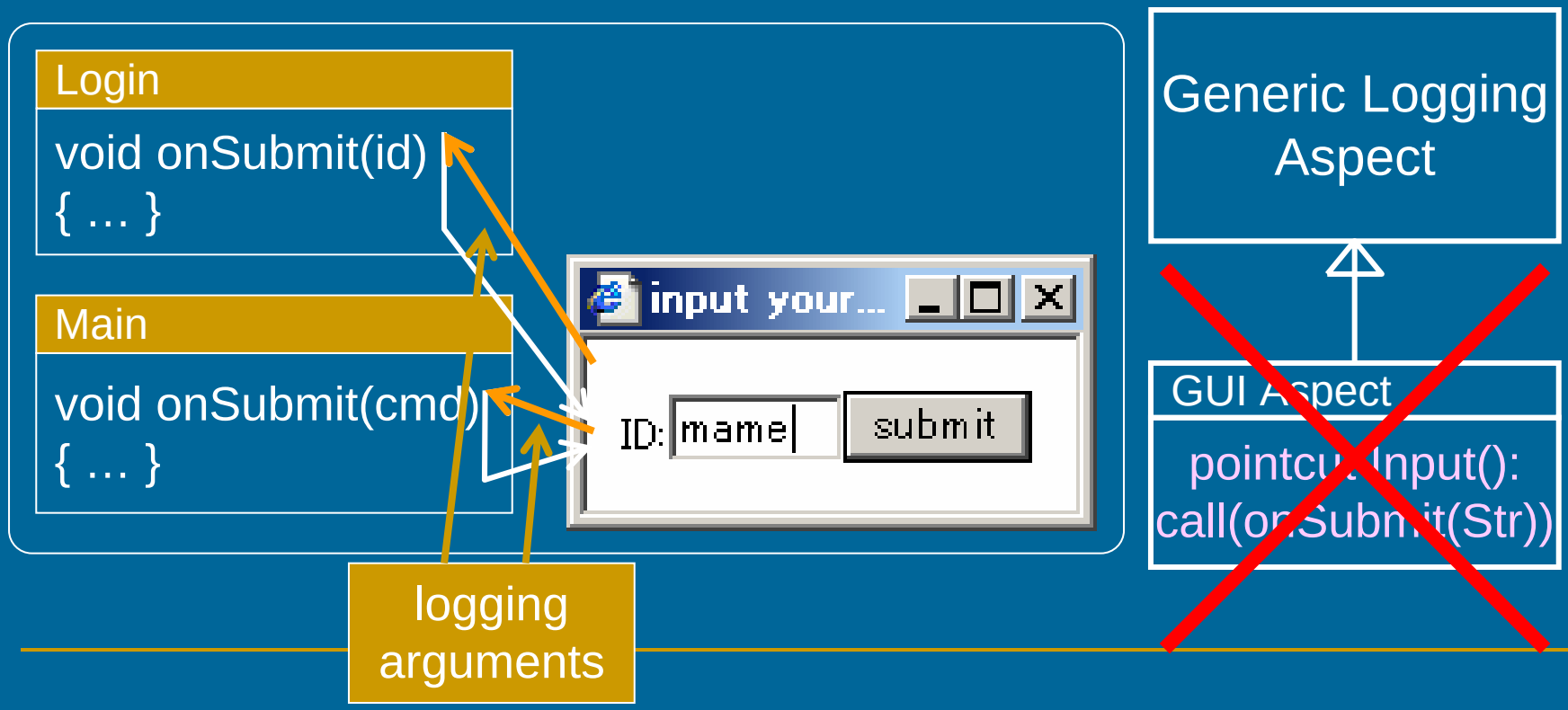
A. Merely concretize an aspect additionally

Aspect reusability



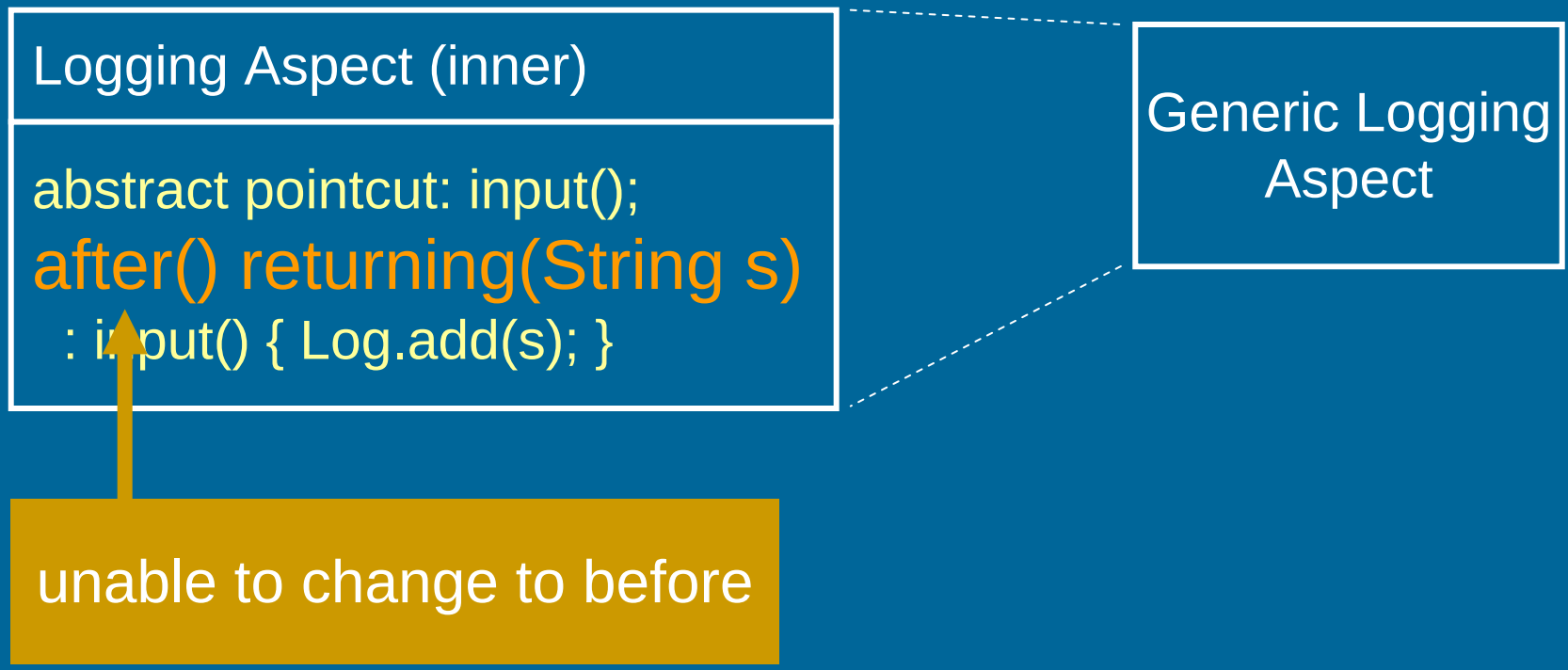
Problem: Aspects are not as reusable as expected

- Example: A generic logging aspect
  - can **NOT** log inputs in a **GUI** program by defining a pointcut



Why can’t we reuse the aspect?

- Timing of advice execution depends on both **advice modifiers** and pointcuts



Workaround in AspectJ is awkward: overview

- Required changes for more reusable aspect:
  - generic aspect (e.g., logging)
    - two abstract pointcuts, two advice decls. and an auxiliary method
  - concrete aspects
    - two concrete pointcuts even if they are not needed

Workaround in AspectJ is awkward: how to define generic aspect

1. define two pointcuts for before and after
2. define two advice decls for before and after
3. define auxiliary method

Simple Logging Aspect

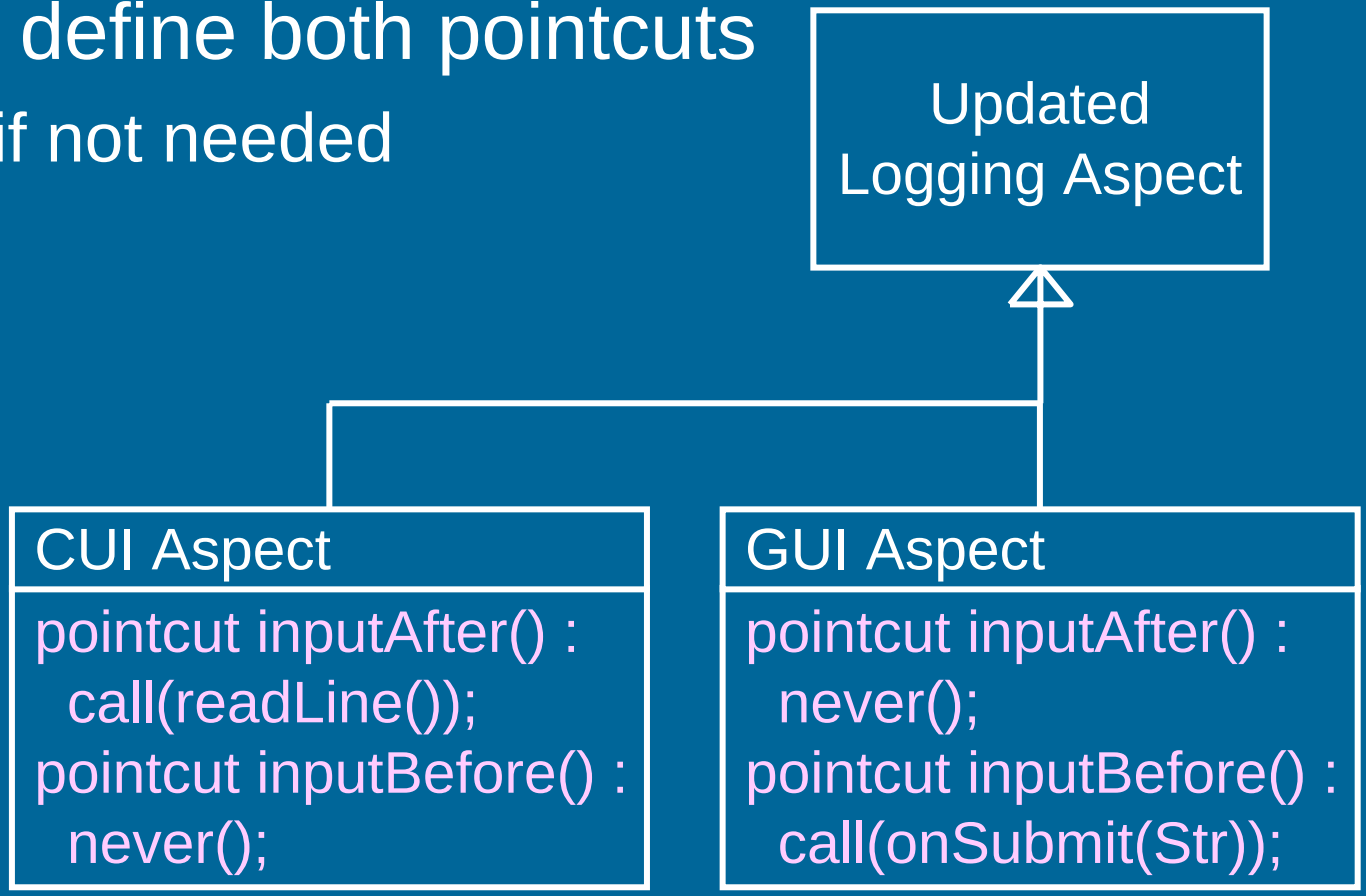
```
abstract pointcut: inputAfter();
abstract pointcut: inputBefore();

after() returning(String s)
  : inputAfter() { log(s); }
before(String s)
  : inputBefore() && args(s)
  { log(s); }

void log(String s) { Log.add(s); }
```

Workaround in AspectJ is awkward: how to define concrete aspects

- always define both pointcuts
  - even if not needed



Summary: Aspect Reusability Problem

- Aspects are not reusable
  when advice modifiers need to be changed
  - CUI/GUI is not an artificial example
    - stand-alone ⇔ application framework
    - blocking I/O ⇔ non-blocking I/O
- Workaround is awkward

- Cause: Timing of advice execution depends on both **advice modifiers** and pointcuts

Contributions

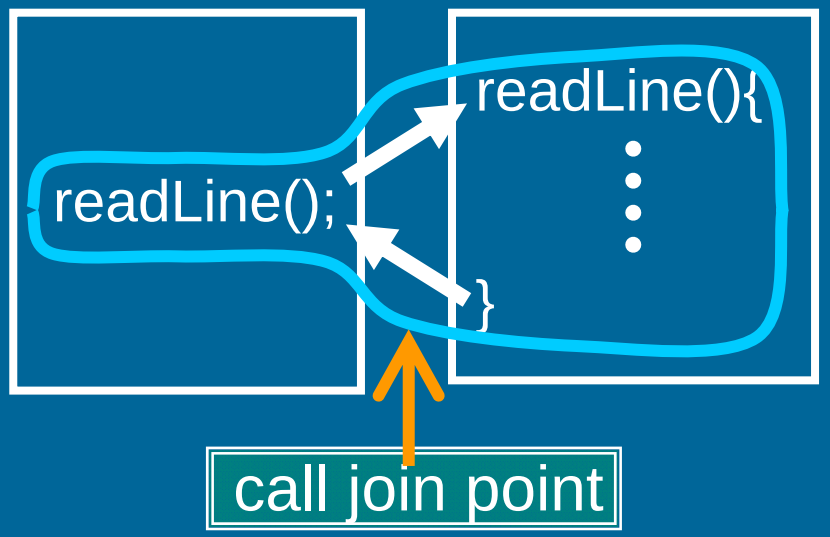
- The point-in-time join point model
- PitJ: an experimental AOP language based on the model
  - completed the language design
- Pitλ: simplified version of PitJ based on λ-calculus
  - a working interpreter
  - formalized in CPS

Point-in-Time Join Point Model

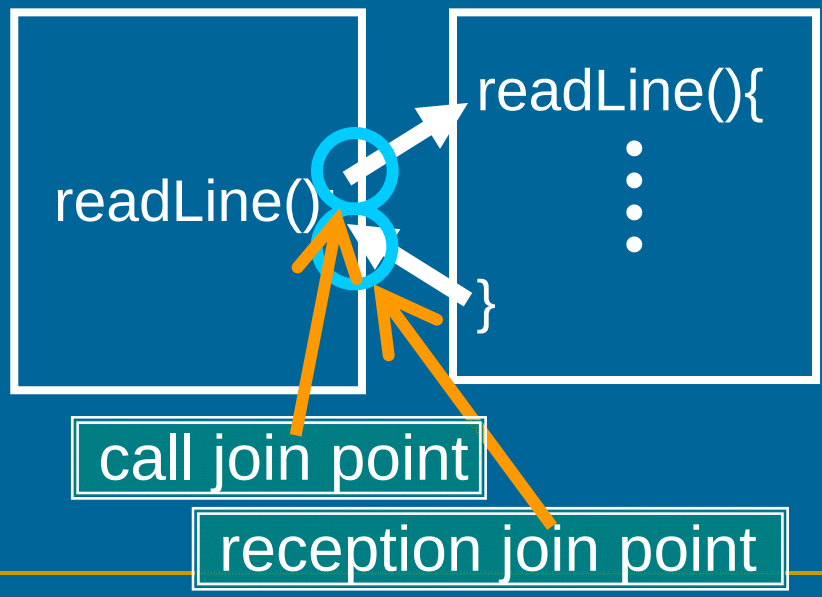
- Define ends of actions as different join points from beginnings of actions

region-in-time model (traditional)

AspectJ, AspectWerkz, JBoss AOP, …



point-in-time model (proposed)



PitJ: An Experimental AOP Language Based on Point-in-Time Model

- is more reusable than AspectJ because of point-in-time model
- is as expressive as AspectJ

- base language : Java (AspectJ-like)

PitJ: Pointcuts

- call(*method*): a call to *method*
- reception(*method*): a return from *method*
- failure(*method*): an exceptional return from *method*
  - i.e., exception is thrown by *method*
- args(*var*): binding join point's value to *var*
  - call join point's value : argument
  - reception join point's value : return value
  - failure join point's value : exception object

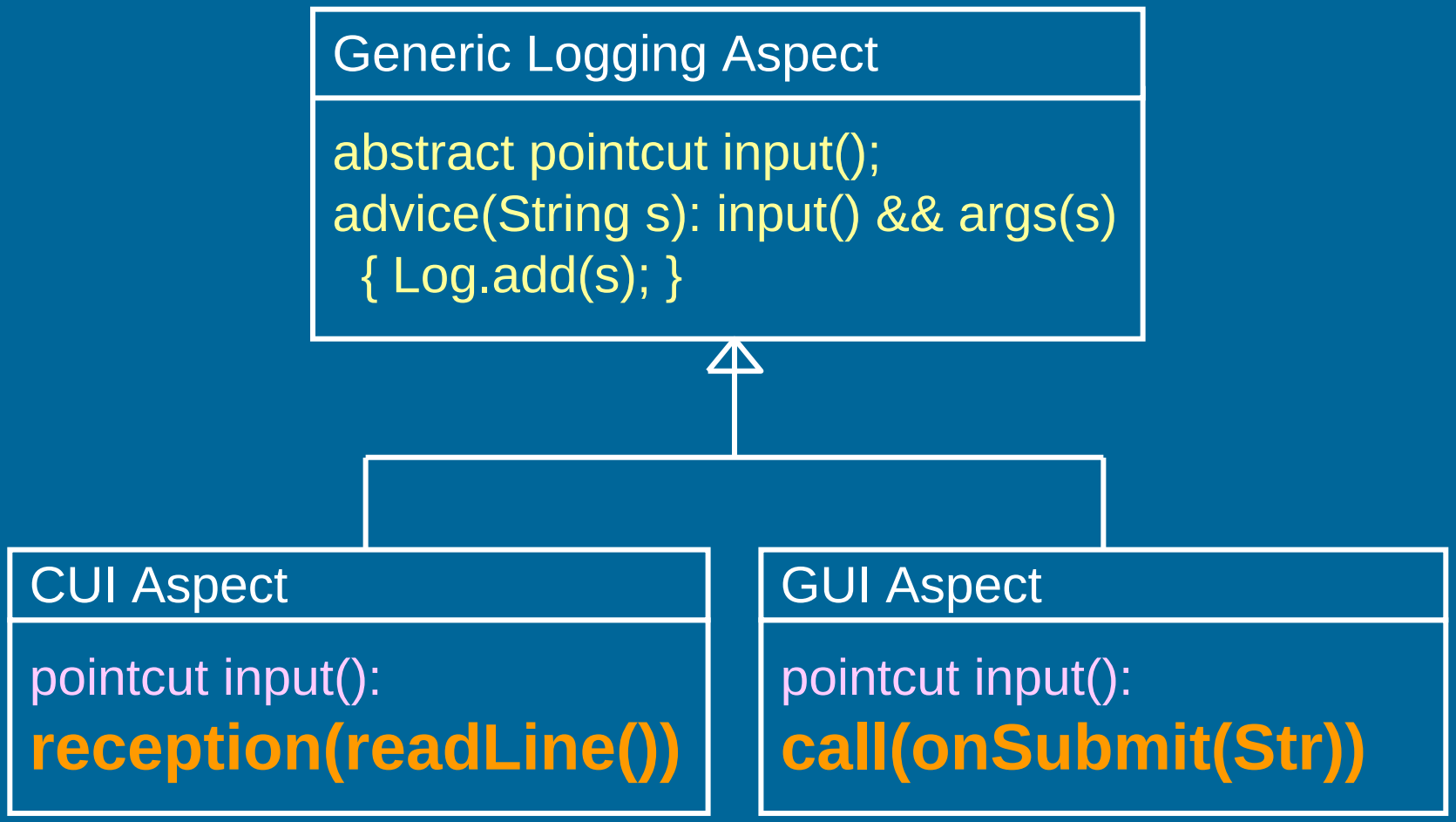
PitJ: Examples of Advice (1)

- No need for advice modifiers
  - advice(Str s): call(m) && args(s) { … }
    - advices at call join point of the method m
    - in AspectJ: before(): call(m) { … }
  - advice(Str s): reception(m) && args(s) { … }
    - in AspectJ: after() returning(Str s): call(m) { … }
  - advice(Obj e): failure(m) && args(e) { … }
    - in AspectJ: after() throwing(Obj e): call(m) { … }

PitJ: Examples of Advice (2)

- before and after advice can be defined in one advice declaration
  - advice(Str s):
    (call(onSubmit(Str)) || reception(readLine())) &&
    args(s) { … }
    - runs at both call join point of onSubmit and a reception join point of readLine
    - in AspectJ, corresponding to a pair of advice decls.
    - before(String s): call(onSubmit(Str)) && args(s) { … }
    - after() returning(String s): call(readLine()) { … }

Reusable Logging Aspect in PitJ



PitJ: Around-like Advice

- usages of around advice in AspectJ
  1. replace the parameters to a join point with new ones
  2. replace the return value to the caller of a join point
  3. go back to the caller without executing a join point
  4. execute a join point more than once
- In PitJ, these are realized by:
  - 1, 2 → return in advice body
  - 3 → new construct: skip
  - 4 → special function: proceed

return in advice body (1)

- replaces join point's value
- Example: at call join point
  - advice(Str s): call(m) && args(s) { **return sanitize(s);** }
    replaces the argument of m with the sanitized one
  - in AspectJ:
    - around(Str s): call(m) && args(s)
      { return proceed(sanitize(s)); }

return in advice body (2)

- Example: at reception join point
  - advice(Str s): reception(m) && args(s)
    { return sanitize(s); }
    replaces the return value of m with the sanitized one
  - in AspectJ:
    - around(Str s): call(m) && args(s)
      { return sanitize(proceed(s)); }

new construct: skip

- skip is evaluated in a call join point:
  - skips subsequent advice decls. and the call itself
    - i.e., jumps to the corresponding reception join point
- in a reception or failure join point:
  - skips subsequent advice decls.
- Example:
  - advice(): call(readLine()) { **skip “dummy”;** }
    makes readLine always return “dummy”
  - in AspectJ:
    - String around(): call(readLine()) { return “dummy”; }

special function: proceed

- proceed is evaluated in a call join point:
  - executes the action until the corresponding reception join point
- in a reception or failure join point:
  - no effect

- Example:
  - advice(): call(readLine) { **proceed();** }
    - let readLine skip every other line
  - advice(): call(readLine) { **skip(proceed() + proceed());** }
    - let readLine return a concatenation of two lines
  - advice(): call(readLine) { **skip(proceed());** }
    - no effect

Summary: PitJ

- No need for advice modifiers
- Advice decls. are more reusable than AspectJ's due to the point-in-time model
- PitJ is as expressive as AspectJ's advice mechanism
  - before : call join points
  - after : reception or failure join point
  - around-like : skip and proceed

Formalization of Point-in-Time Model

- target: Pitλ
  - simplified version of PitJ
  - base language: untyped λ-calculus
- approach:
  - denotational semantics in continuation-passing style
  - key idea: denote **join points** as **applications to continuation**

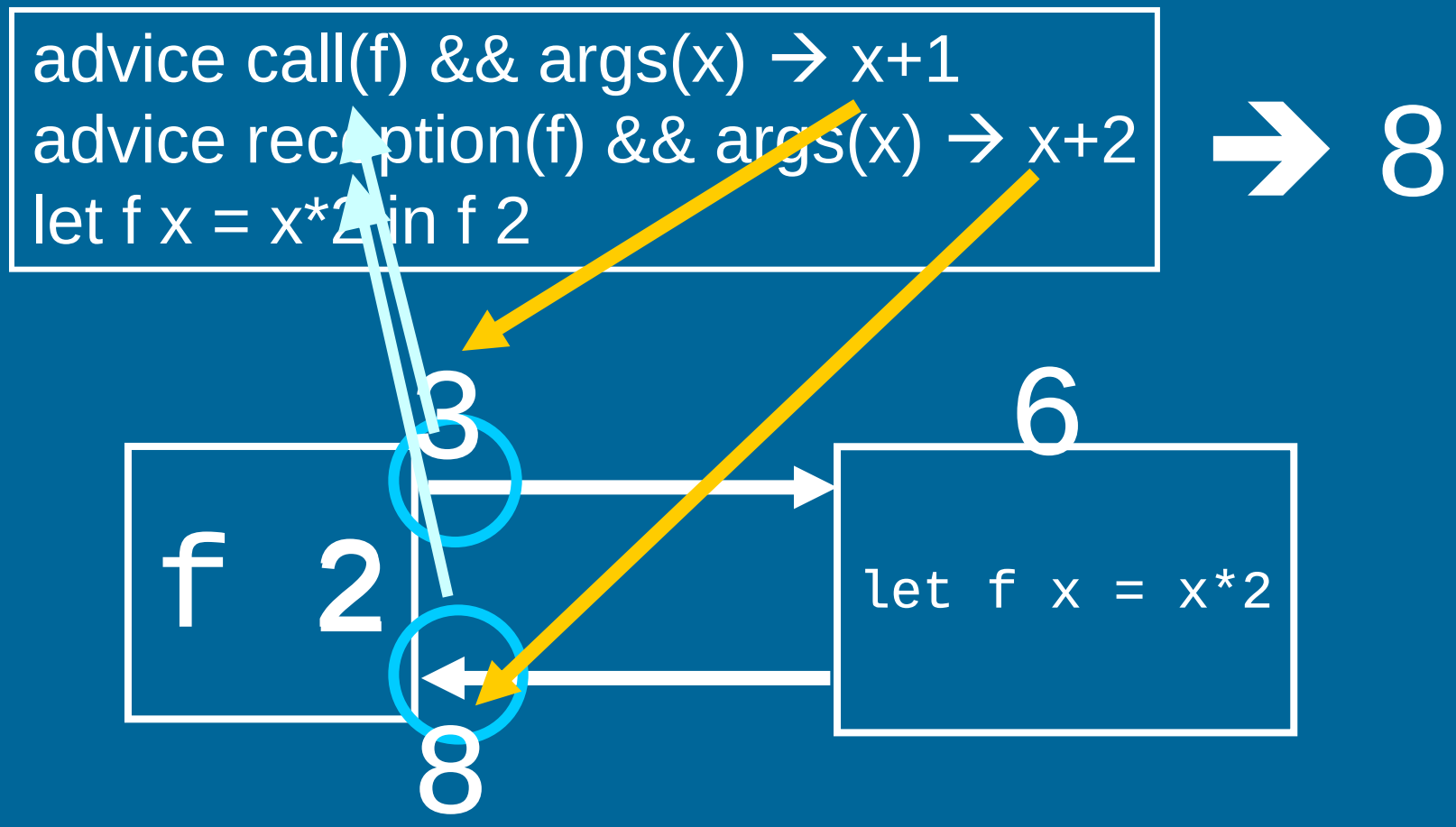
Semantic Equations: Advice

- $\mathcal{A}$: advice list → Event → Ctn → Ctn
  - Event : kind of join point
  - Ctn : continuation
- $\mathcal{A}$ [A] ε κ: return continuation that:
  - selects applicable advice decls. from A (list of advice)
  - executes them, and
  - executes κ (continuation)
    - ε: kind of join point

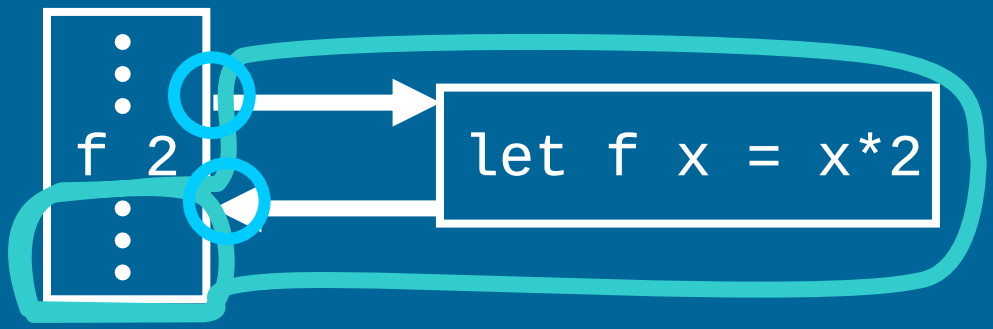
Semantic Equations: Expression

- $\mathcal{E}$: expression → Ctn → Ans
  - Ctn : continuation
  - Ans : answer
- $\mathcal{E}$ [E] κ: evaluates E and executes κ
  - E : expression
  - κ: continuation

Sample Program in Pitλ



Semantics of Function Call (abridged)



semantics of λ-calculus with an abstract mechanism

$$\mathcal{E}[\![\mathsf{E}_0\ \mathsf{E}_1]\!]\,\kappa = \mathcal{E}[\![\mathsf{E}_0]\!]\,(\lambda f.\ \mathcal{E}[\![\mathsf{E}_1]\!]\,(\lambda v.\ \mathcal{A}[\![\mathbf{A}]\!]\ \mathrm{call}\ (f\ (\lambda v.\mathcal{A}[\![\mathbf{A}]\!]\ \mathrm{reception}\ \kappa\ v))\ v))$$

application ← application → continuation

= reception join point = call join point

we can define it in systematic way!

Advantages of Our Formalization

- simpler than existing formalizations [Wand ’02] [Walker ’03]
  - no need for rules for each advice modifier
  - beginnings and ends of actions are represented symmetrically
- easier to support advanced features
  - exception handling
  - context sensitive pointcuts (cflow)
  - around advice

exception handling (sketch)

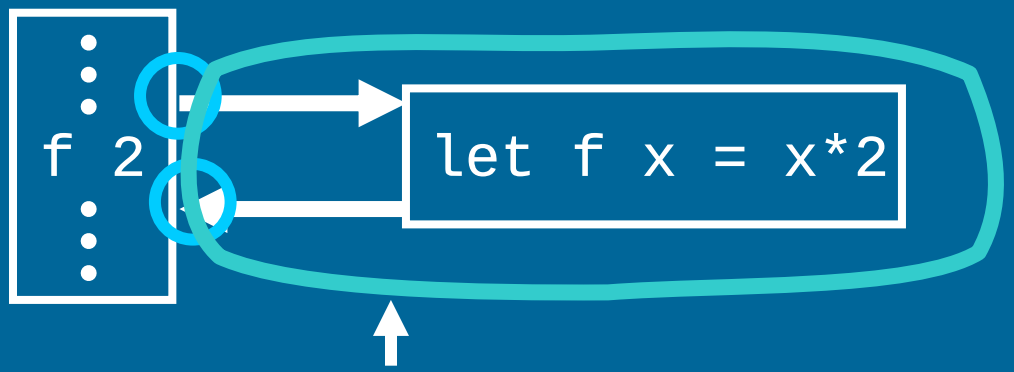
- give a standard semantics
  - by adding continuation that represents current handler
- identify failure join point

semantics of λ-calculus **with** advice mechanism

$$\mathcal{E}[\![\mathsf{E}_0\ \mathsf{E}_1]\!]\ \kappa\ \kappa_h = \mathcal{E}[\![\mathsf{E}_0]\!]\ (\lambda f.\ \mathcal{E}[\![\mathsf{E}_1]\!]\ (\lambda v.\ \mathcal{A}[\![\mathsf{A}]\!]\ \text{call}\ (f\ (\lambda v.\ \mathcal{A}[\![\mathsf{A}]\!]\ \text{reception}\ \kappa\ \kappa_h\ v)\ \kappa_h\ v)\ \kappa_h\ (\lambda v.\ \mathcal{A}[\![\mathsf{A}]\!]\ \text{failure}\ \kappa_h\ \kappa_h\ \kappa_h\ v)\ \kappa_h\ v))$$

around-like advice (concept)

- using idea of partial continuation [Danvy '89]
  - a part of the rest of computation, rather than the whole rest



- we currently formalized by using continuation-composing style

Related Work

- approaches based on the region-in-time model:
  - Aspect SandBox[Wand '02], Tucker et al. '03, MiniMAO[Clifton '05],
- some approaches treat beginning and end of an event as different join points, but that have different motivations
  - Walker et al. '03: propose a low-level language that serves as a target of translation from a high-level AOP language
  - Douence et al. '04: define a formal semantics of cflow by using calling contexts from execution history

Conclusion

- a new join point model that defines beginnings and ends of actions as different join points
  - Point-in-time vs. Region-in-time
  - designed PitJ based on the model
    - improves aspect reusability by enhancing expressiveness of pointcuts
  - formalized the model in continuation-passing style
    - simpler than some existing formalizations
    - easier to support advanced features

Future Work

- integrate more advanced features
  - dflow pointcut [Kawauchi '03]
  - first-class continuation
  - tail-call elimination
- implement a compiler for PitJ language
  - Java bytecode should be made without CPS transformation