# Is Join Point a Point?

## a pointcut and advice mechanism for making aspects more reusable

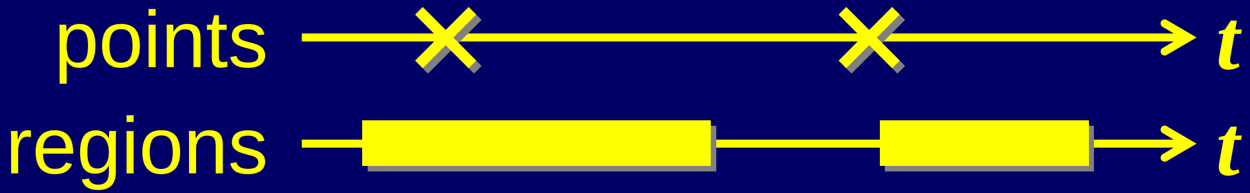
Hidehiko Masuhara
University of Tokyo

joint work with Yusuke Endoh and Aki Yonezawa

## A secret of AspectJ

- AspectJ is based on ***join points***
- A sceret of AspectJ:

  join points are ***not*** points, but ***regions***

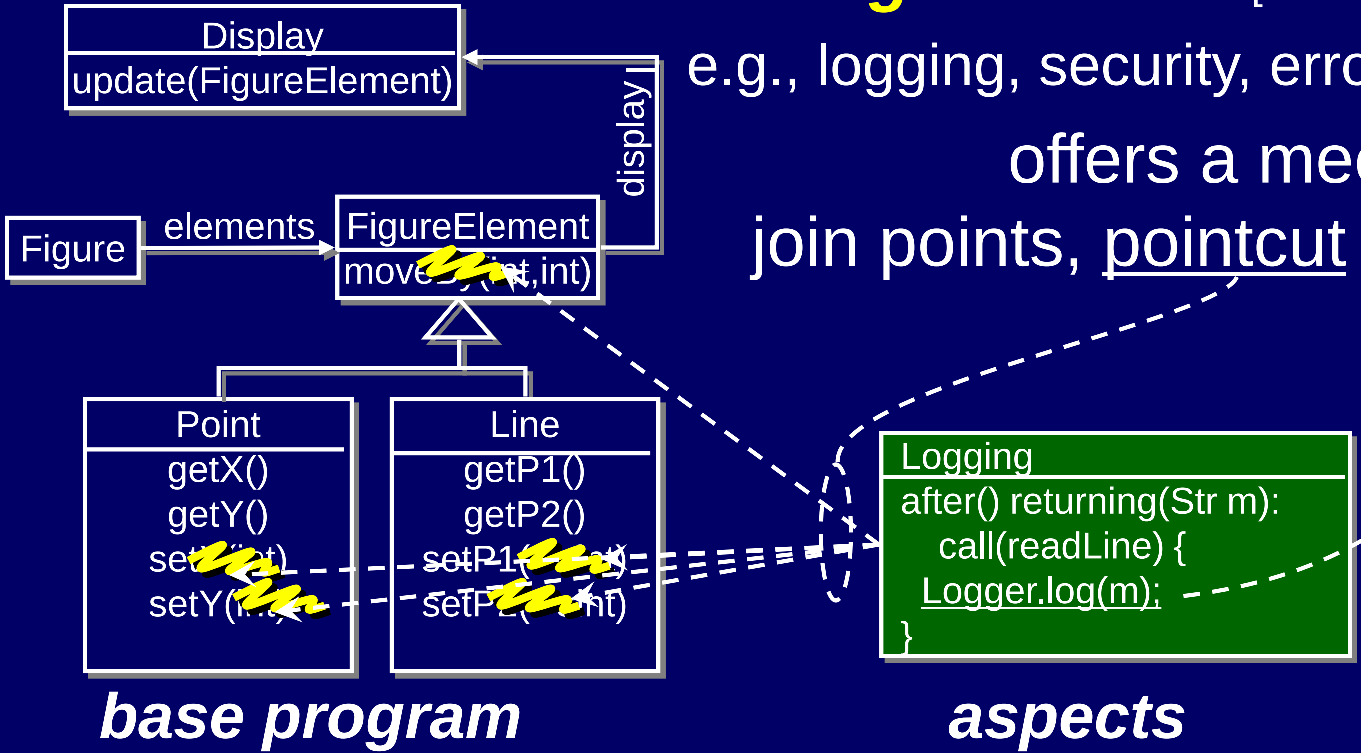


  – makes aspects hard to maintain
- We propose an AOPL
  in which join points are ***points***

# Aspect-oriented programming

enables modularization of
***crosscutting concerns*** [Kiczales et al.1997]
e.g., logging, security, error handling

offers a mechanism:
join points, pointcut & advice



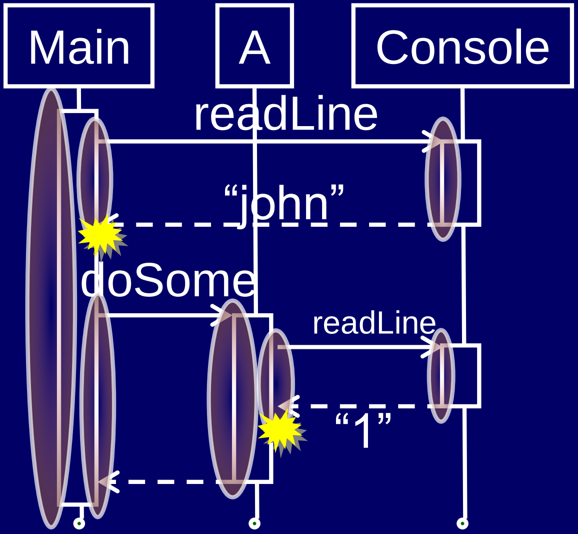
# Join point, pointcut & advice in AspectJ

Base: reads user's inputs in several modules

Logging: record all inputs from the console

- Join points=actions like method calls, execs...
- Pointcut: selects jps
- Advice: runs on selected jps

```
after() returning(String x) :
    call(String *.readLine()) {
 Logger.log(x);
}
```



## Aspect maintainability: most changes are adapted by pointcuts

Changes in aspect spec./base prog. ↓

```
after() returning(String x) :
        (call(String *.readLn())
       || call(String *.getenv()))
       && !within(LogBrowser) {
  Logger.log(x);
}
```

???

- log getenv as well
- exclude calls from LogBrowser
- rename readLine to readLn
- log onSubmit as well

↑modifications to pointcuts
to cope with changes

## Aspect maintainability: some changes can be adapted by pointcuts

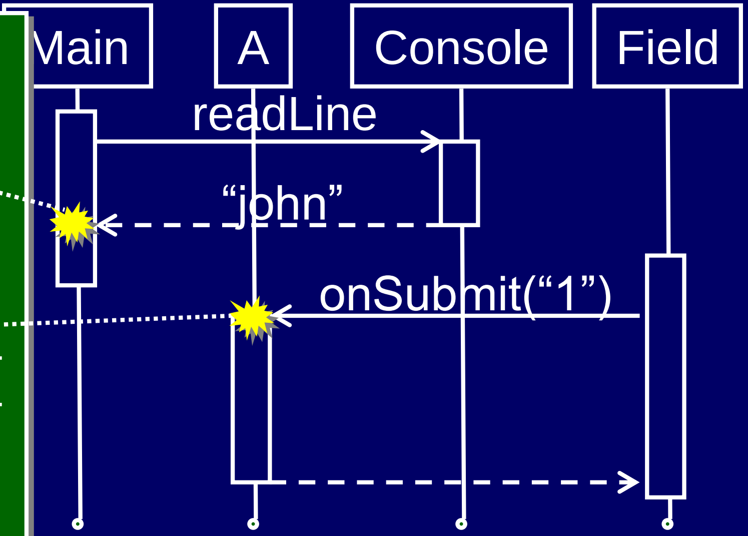
advice specifiers
(not pointcuts)

Logging inputs from console & GUI widgets

needs two advice decls.
i.e., ***can not be adapted by pointcuts***

provide inputs thru callback methods

```
after() returning(String x) :
    call(String *.readLine()) {
  Logger.log(x);
}
before(String x):
    exec(* *.onSubmit(String)) && args(x) {
  Logger.log(x);
}
```



# Another example: returning null vs. throwing exceptions

```
r = find(...);
if (r==null)
   handle not found case
process the result
```

```
after()returning(Result r):
   call(find)&&if(r==null) {
   Logger.log();
}
```

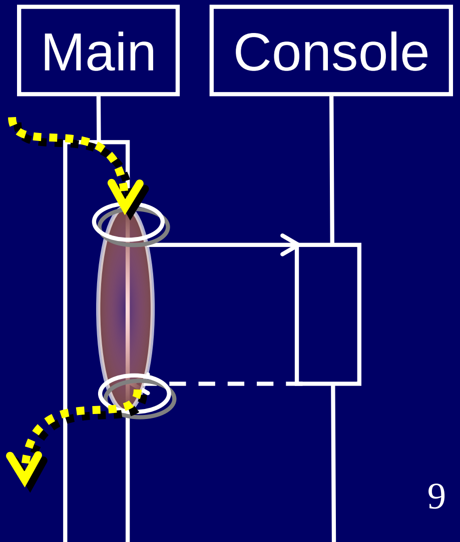
---

```
try {r = find(...);}
catch (NotFound e) {
   handle not found case
}
process the result
```

```
after()throwing(NotFound):
   call(find) {
   Logger.log();
}
```

## Problem summary & analysis

- Generalization: can not advise "beginnings of X and ends of Y" by one decl.
  - active / passive parameter passing
  - returning error values / throwing exceptions
  - direct style / continuation passing style (in FPL)
- Reasons:
  - join points are ***regions*** w/ entry and exit
  - pointcuts select only join points
  - advice decls. specify entry or exit



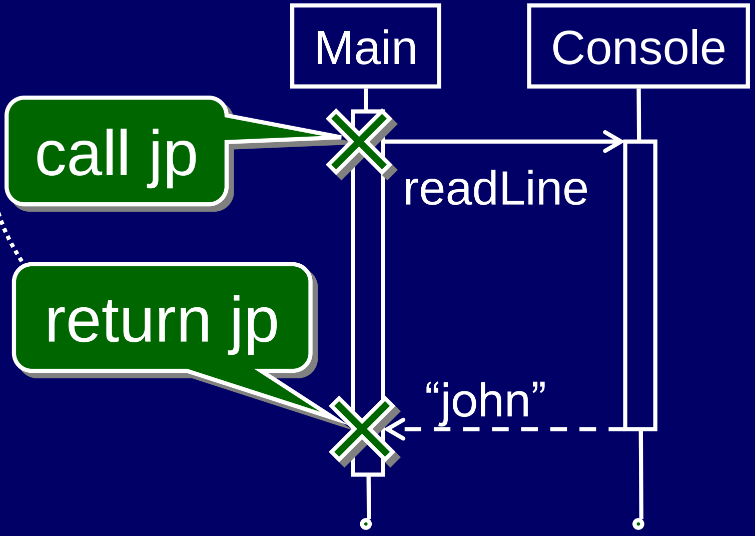
## Proposal: AOP mechanism based on ***point-in-time*** join points

- Overview
- Aspect maintainability
  with point-in-time join points
- Design issues of pointcuts and advice
- Formalization

## AOP mechanism based on ***point-in-time*** join points

- A join point is a point in time
- New join points that represent ends of actions
- New pointcuts that select new join points

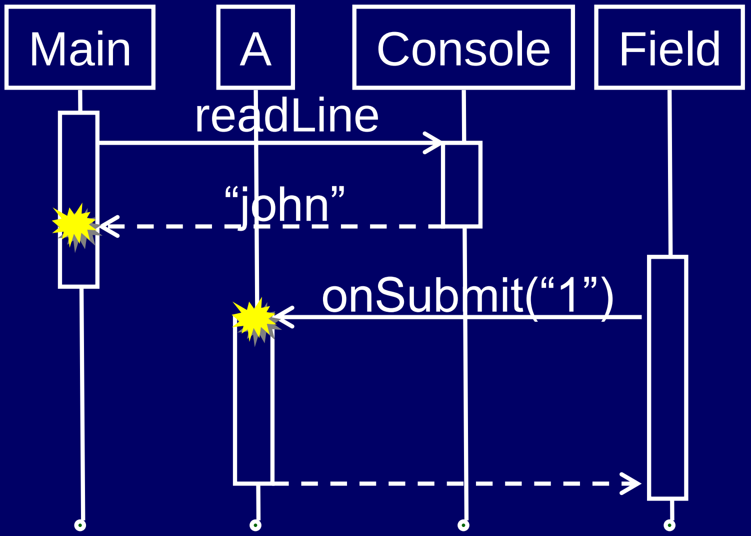
```
advice(String x) :
        return(String *.readLine())
        && args(x) {
  Logger.log(x); proceed x;
}
```



# Aspect maintainability with point-in-time join points: *logging readLine & onSubmit*

- One advice decl. can log both
  - return values from readLine
  - parameters to onSubmit

```
advice(String x) :
   (return(String *.readLine())
   || call(void *.onSubmit(String x)))
   && args(x) {
  Logger.log(x); proceed x;
}
```



# Aspect maintainability with point-in-time join points: *returning null vs. throwing exceptions*

```
r = find(...);
if (r==null)
   handle not found case
do with the result
```

```
try {r = find(...);}
catch (NotFound e) {
   handle not found case
}
do with the result
```

advice(): (***return***(find) &&
   args(r) && if(r==null)) ||
   ***throw***(find,NotFound) {
   Logger.log();proceed;
}

one advice decl.
for two

# Design issues of pointcuts & advice with point-in-time join points

convert parameters to onSubmit into lowercase

introduced a special

proceed to a return jp

can

introduced another form to skip to the caller

```
after() returning(String x) :
```

```
void around(String x) :
```

```
String around() :
```

ver();

-time

```
String around() :
    call(readLine()) {
  return "dummy";
}
```

region-in-time

))

advice(String x) :

g))

jp

re and af

```
advice() : call(readLine()) {
  skip "dummy";
}
```

point-in-time

point-in-time

# Design issues of pointcuts & advice with point-in-time join points

- ost features in AspectJ

  a

  j

  o

```
void around():
     call(*.onSubmit()) {
  int start=getTime();
  proceed();
  print(getTime()-start);
}
```

region-in-time

```
String around():
     call(*.readLine()) {
  return proceed()+proceed();
}
```

region-in-time

- ...but difficult to support some:
  - repeat execution of a jp
  - run code before **and** after a jp

proceed won’t come back in point-in-time

## Formalization of pointcut & advice based on point-in-time join points

Writing a denotational semantics

- of an untyped FPL + pointcut&advice
- by using a continuation passing style (CPS)
  - a return = application to a continuation
- simpler in terms of advice exec.
  - no longer has specifiers like “before”
- suitable to explore advanced features
  - e.g., advising exceptions

# Semantics of advice execution: a sample session

An expression:

```
let f(x)=x+x in f(1)
```

with advice:

```
advice(x):call(f){
   proceed x+1;
}
advice(x):return(f){
   proceed x/2;
}
```

Execution trace:

1. creates a jp "call f with 1"
2. matches pointcut "call(f)"
3. evaluates "proceed x+1"
4. calls f with 2
5. creates a jp
   "return from f with 4"
6. matches pointcut
   "return(f)"
7. evaluates "proceed x/2"
8. yields 2

## Semantics of advice execution: function call w/o advice

- semantic function

$\mathcal{E} : \mathit{Exp} \to \mathit{Env} \to \mathit{Ctn} \to \mathit{Ans}$

$\mathit{Ctn} = \mathit{Val} \to \mathit{Ans}$

$$\mathcal{E}[(E_0\ E_1)]\rho\ \kappa = \mathcal{E}[E_0]\rho\ (\lambda f.\ \mathcal{E}[E_1]\ \rho\ (\lambda v.\ \boxed{f\ (\lambda v'.\ \boxed{\kappa\ v'})\ v}))$$

return

call

a function is denoted by a term of type $\mathit{Ctn} \to \mathit{Val} \to \mathit{Ans}$

# Semantics of advice execution: function call **with** advice

- semantic function

$$\mathcal{E} : \mathit{Exp} \to \mathit{Env} \to \mathit{Ctn} \to \mathit{Ans}$$

$$\mathit{Ctn} = \mathit{Val} \to \mathit{Ans}$$

$$\mathcal{E}[(E_0\ \ E_1)]\,\rho\ \kappa\ = \mathcal{E}[E_0]\,\rho$$
$$(\lambda f.\ \mathcal{E}[E_1]\ \rho\ (\lambda v.\ \boxed{\mathcal{W} A\ \theta \boxed{\phantom{x}} v}\ ))$$

$$\boxed{\lambda v'.\ \mathcal{W} A\ \theta'\ \kappa\ v'}$$

can treat call & return jps uniformly

jp "call f"

weaver

advice decls.

jp "return f"

# Semantics of advice execution: weaver

- $\mathcal{W}$: $Adv \to Jp \to Ctn \to Ctn$

```
𝒲 [advice(x): p {E}] θκv =
   if p matchesθ
       then ℰ [E] [v/x] κ
       elseκ v
```

# Semantics of advanced features (ongoing)

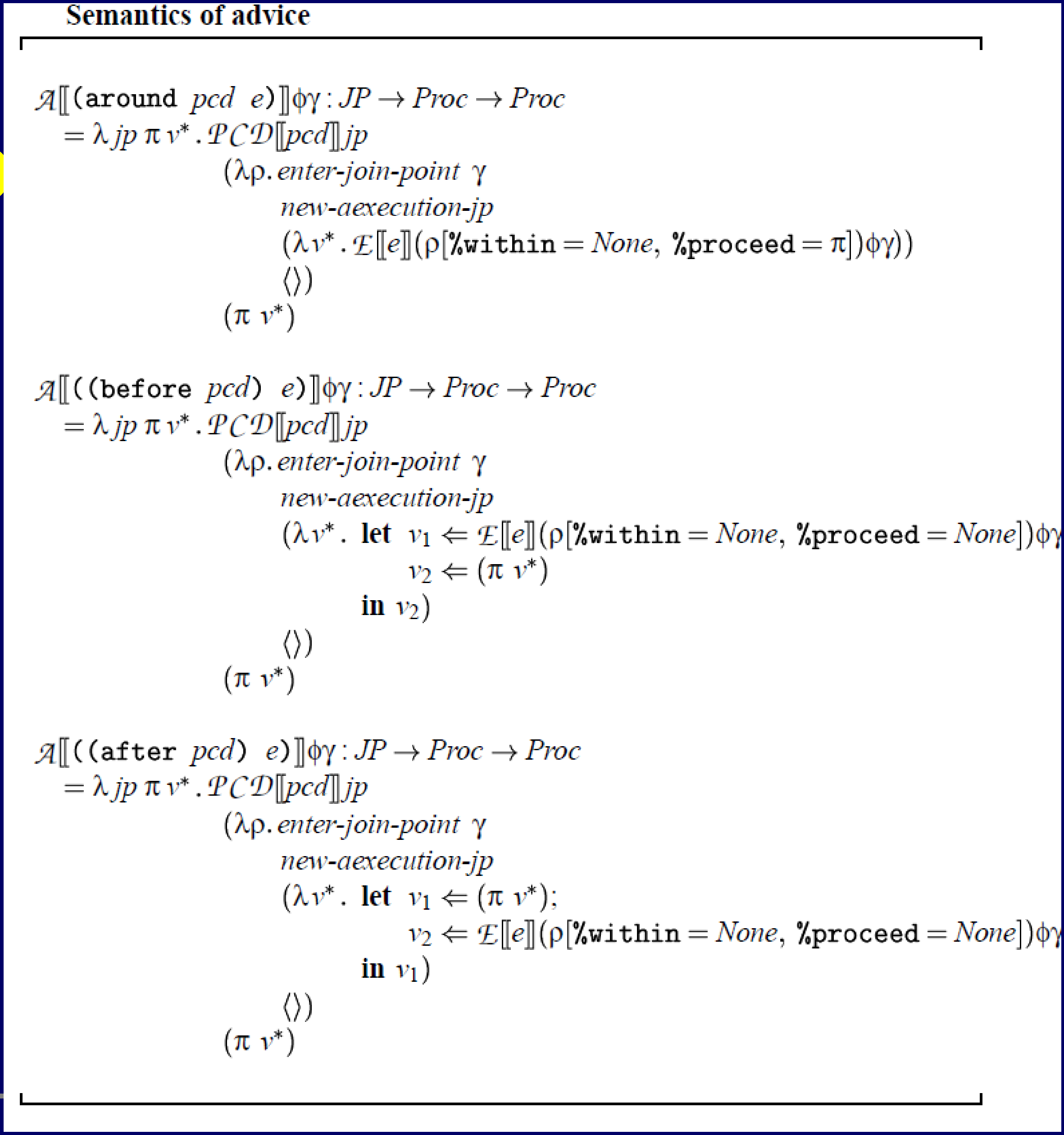
- Uniform representation of ***exception throwing*** mechanisms
  - represents exception handlers as continuations
  - creates “throw” join point at throwing exceptions
- Support for ***history sensitive pointcuts***
  - similar approach to tracecuts [Walker00]
  - would subsume cflow
- Interaction with ***tail call elimination***
  - crucial in FPL
  - folding eta-expanded continuations

# Related work: extension to pointcuts and advice

- Poincuts that capture return values:
  dflow [APLAS'03], Arachne [Douence'05]
  - based on region-in-time join points
- Fine grained jps:
  LoopsAJ[Harbulot'05], Eos-T[Rajan'05], bugdel[Usui'05]
  - based on region-in-time join points

## Related wo

- Aspect SandBox [Wand'02]
  - region-in-time, denotational & direct style
  - semantic function for each of before/after/around



**Semantics of advice**

$$
\begin{array}{l}
\mathcal{A}[\![(\texttt{around}\ pcd\ \ e)]\!]\phi\gamma : JP \to Proc \to Proc \\
\quad = \lambda\, jp\ \pi\ v^* .\, \mathcal{PCD}[\![pcd]\!]jp \\
\qquad\qquad\qquad (\lambda\rho .\, \textit{enter-join-point}\ \ \gamma \\
\qquad\qquad\qquad\quad \textit{new-aexecution-jp} \\
\qquad\qquad\qquad\quad (\lambda v^* .\, \mathcal{E}[\![e]\!](\rho[\texttt{\%within} = None,\ \texttt{\%proceed} = \pi])\phi\gamma)) \\
\qquad\qquad\qquad\quad \langle\rangle) \\
\qquad\qquad\qquad (\pi\ \ v^*)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{A}[\![((\texttt{before}\ \ pcd)\ \ e)]\!]\phi\gamma : JP \to Proc \to Proc \\
\quad = \lambda\, jp\ \pi\ v^* .\, \mathcal{PCD}[\![pcd]\!]jp \\
\qquad\qquad\qquad (\lambda\rho .\, \textit{enter-join-point}\ \ \gamma \\
\qquad\qquad\qquad\quad \textit{new-aexecution-jp} \\
\qquad\qquad\qquad\quad (\lambda v^* .\ \ \textbf{let}\ \ v_1 \Leftarrow \mathcal{E}[\![e]\!](\rho[\texttt{\%within} = None,\ \texttt{\%proceed} = None])\phi\gamma \\
\qquad\qquad\qquad\qquad\qquad\qquad v_2 \Leftarrow (\pi\ \ v^*) \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ v_2) \\
\qquad\qquad\qquad\quad \langle\rangle) \\
\qquad\qquad\qquad (\pi\ \ v^*)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{A}[\![((\texttt{after}\ \ pcd)\ \ e)]\!]\phi\gamma : JP \to Proc \to Proc \\
\quad = \lambda\, jp\ \pi\ v^* .\, \mathcal{PCD}[\![pcd]\!]jp \\
\qquad\qquad\qquad (\lambda\rho .\, \textit{enter-join-point}\ \ \gamma \\
\qquad\qquad\qquad\quad \textit{new-aexecution-jp} \\
\qquad\qquad\qquad\quad (\lambda v^* .\ \ \textbf{let}\ \ v_1 \Leftarrow (\pi\ \ v^*); \\
\qquad\qquad\qquad\qquad\qquad\qquad v_2 \Leftarrow \mathcal{E}[\![e]\!](\rho[\texttt{\%within} = None,\ \texttt{\%proceed} = None])\phi\gamma \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ v_1) \\
\qquad\qquad\qquad\quad \langle\rangle) \\
\qquad\qquad\qquad (\pi\ \ v^*)
\end{array}
$$

# Final remarks: a pointcut & advice mechanism based on point-in-time join points

- Design
  - can uniformly treat beginnings and ends of actions
  - some missing features (eg repeating jps)
- Semantics
  - in a continuation passing style
  - uniformly treat calls and returns
  - advanced features (eg exception, cflow, TCE)
- Implementation
  - development of a compilation model
- Evaluation
  - assesment of aspect maintainability

future work