# Aspectual Caml

*an Aspect-Oriented Functional Language*

Hidehiko Masuhara
Hideaki Tatsuzawa
Akinori Yonezawa
University of Tokyo

# Background: Aspect-Oriented Programming

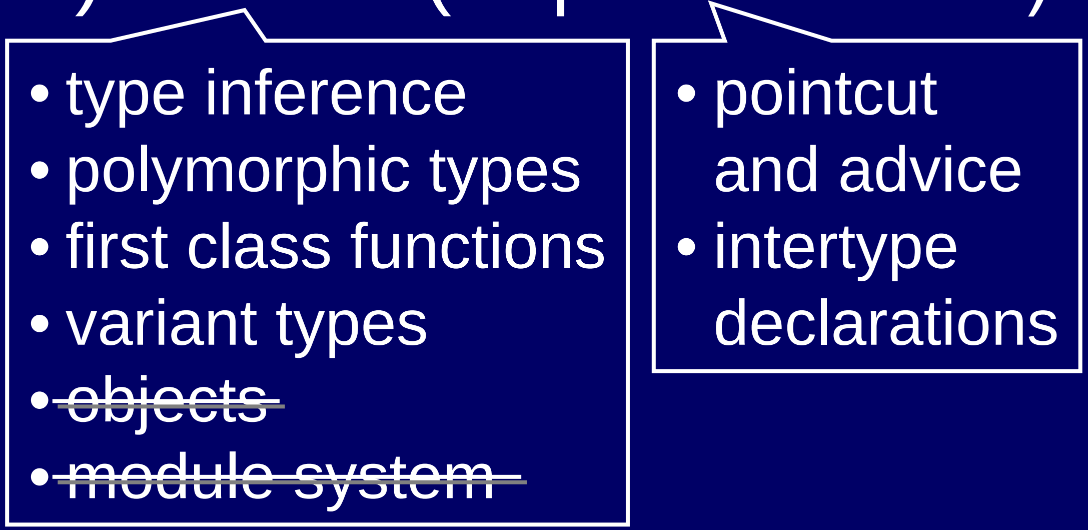
- for separation of crosscutting concerns
  - ***complements*** existing modularization mechanisms like OOP, FP, etc.
  - proven to be useful (e.g., logging & exception handling in middleware [Colyer04], optimizations in OS [Coady01])
- mainly developed/used ***in OOP context***
  - AspectJ, AspectWerkz, JBoss AOP, Spring AOP, AspectC++, AspectS, …

# AOP will also be useful to functional programming!

- FPs also have crosscutting concerns
  - aren’t you bothered by logging code?
- Advanced FP features are great,
  but not always help
  - e.g., polymorphic variants + open recursion
    - great for type safe extension of data structures
    - but not all programs are in that style

# Proposal: ***Aspectual Caml*** (A’Caml)

= (Objective) Caml + (AspectJ – Java)



- type inference
- polymorphic types
- first class functions
- variant types
- ~~objects~~
- ~~module system~~

- pointcut and advice
- intertype declarations

- Approach:
  - – design & implement first (formalize later)
  - – see interactions of features
  - – assume compilable implementation

# Example: **simple calculator (base)** + tracing (aspect)

```
# eval empty (parse “let x=1+2 in x+x”) ;;
- : int = 6
```

```
type term =
| Num of int
| Var of string
| Add of term * term
| Let of string * term * term
```

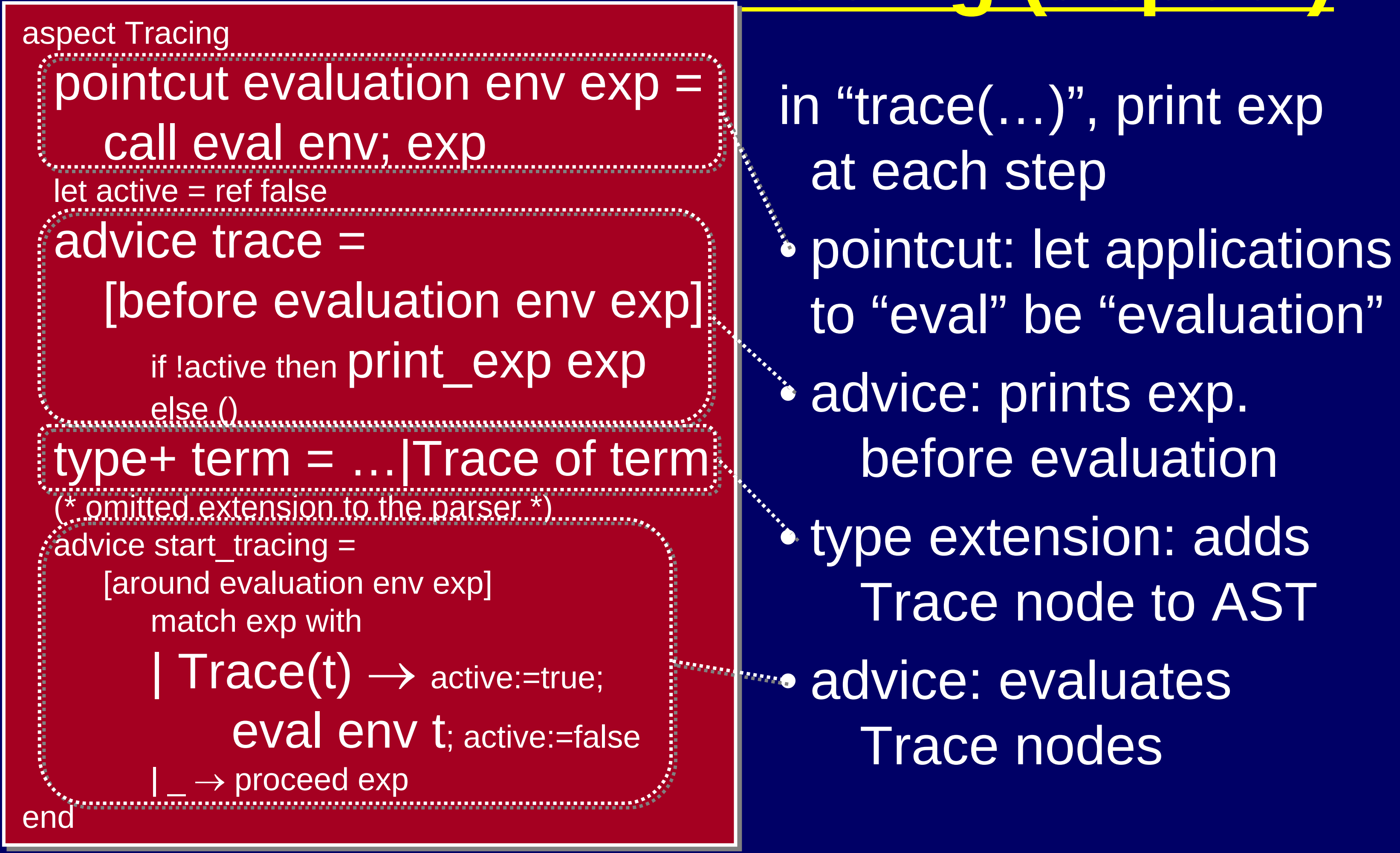
variant type of AST nodes

evaluator

```
let rec eval env exp =
  match exp with
  | Num v → v
  | Var x → lookup env x
  | Add(t1,t2) → let e = eval env in
     (e t1) + (e t2)
  | Let(x,t1,t2) → let v = eval env t1 in
     eval (extend env x v) t2
```

parser

# Example: simple calculator (base) + **tracing (aspect)**

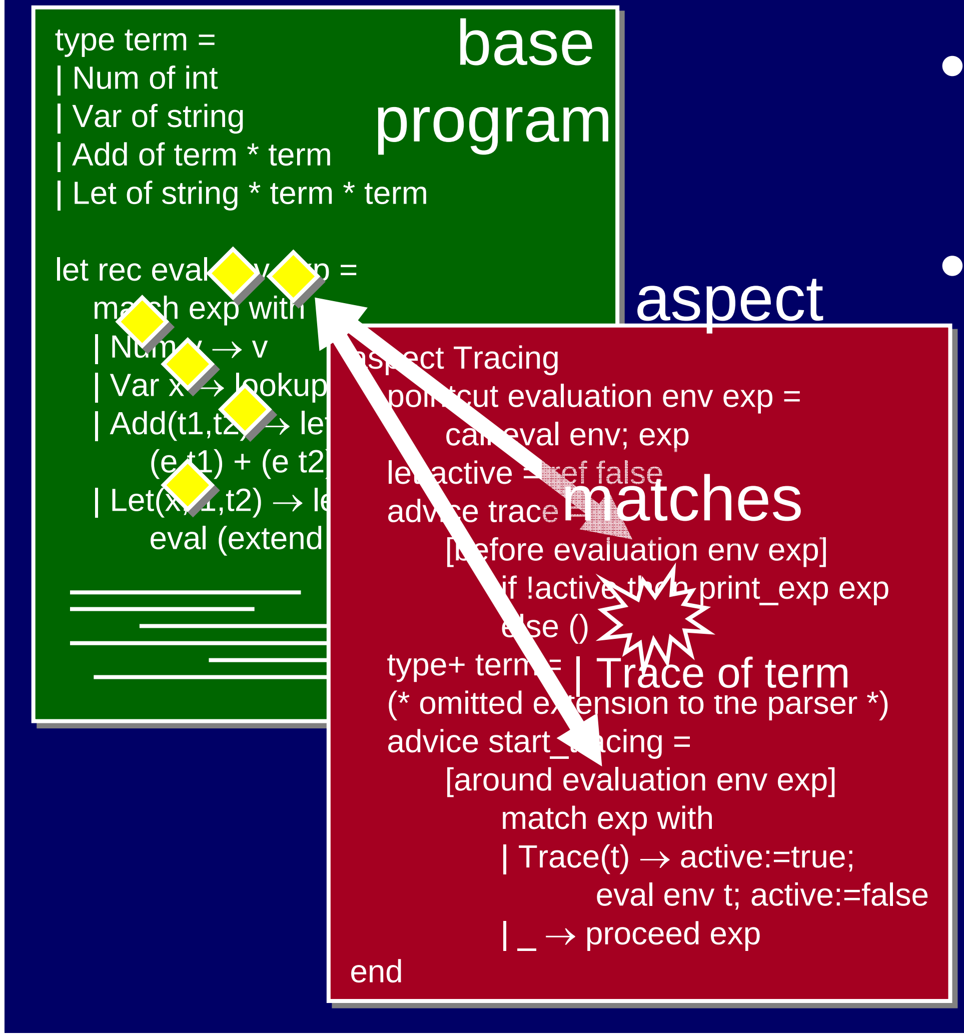


```
aspect Tracing
  pointcut evaluation env exp =
     call eval env; exp
  let active = ref false
  advice trace =
     [before evaluation env exp]
        if !active then print_exp exp
        else ()
  type+ term = …|Trace of term
  (* omitted extension to the parser *)
  advice start_tracing =
     [around evaluation env exp]
        match exp with
        | Trace(t) → active:=true;
             eval env t; active:=false
        | _ → proceed exp
end
```

in “trace(…)”, print exp at each step

- pointcut: let applications to “eval” be “evaluation”
- advice: prints exp. before evaluation
- type extension: adds Trace node to AST
- advice: evaluates Trace nodes

# Execution model of a program with aspects



```
type term =
| Num of int
| Var of string
| Add of term * term
| Let of string * term * term

let rec eval env exp =
   match exp with
   | Num v → v
   | Var x → lookup
   | Add(t1,t2) → let
        (e t1) + (e t2)
   | Let(x,t1,t2) → le
        eval (extend
```

```
aspect Tracing
   pointcut evaluation env exp =
        call eval env; exp
   let active = ref false
   advice trace =
        [before evaluation env exp]
             if !active then print_exp exp
             else ()
   type+ term = | Trace of term
   (* omitted extension to the parser *)
   advice start_tracing =
        [around evaluation env exp]
             match exp with
             | Trace(t) → active:=true;
                       eval env t; active:=false
             | _ → proceed exp
end
```

- Before exec.:
  - extends variant types
- During exec.:
  - generates join point at function application, etc.
  - tests each join point aginst pointcuts
  - runs bodies of matching advice decls.

# How Aspectual Caml adapts AOP features

- Advising curried functions
  - Curried pointcuts
- Type inference in pointcut & advice
  - Type inference before selecting join points
  - Polymorphic / monomorphic pointcuts
- Mechanisms to extend data structure
  - Type extension

# Curried pointcuts: a problem to advise curried functions

- Curried functions are common in FP

```
eval: env → exp → int
...eval env t1...
...let e = eval env in (e t1) + (e t2)...
```

≡ (eval@env)@t1

- AspectJ’s pointcuts capture only first application

```
advice tr = [before call eval r] ...
```

- Solution in Aspectual Caml: ***curried pointcut***

# Curried pointcuts: solution in Aspectual Caml

- Syntax:

```
advice tr = [before call eval r e] ...
```

- Meaning: “applications to the functions that are returned by the 1st application”

```
eval: env → exp → int
...(eval@env) @ t1...
...let e = eval env in (e t1) + (e t2)...
```

- Implementation: ***adivce transformation***

# Curried pointcuts: implementation by advice transformation

Adivce decl. with a curried pointcut:

```
advice tr = [before call eval r e] print_exp e
```

is translated to:

```
advice tr = [around call eval r ]
   let p=proceed r in
      fun e → print_exp e; p e
```

“replace the result of 1st application with a function that runs advice body”

- useful in many situations

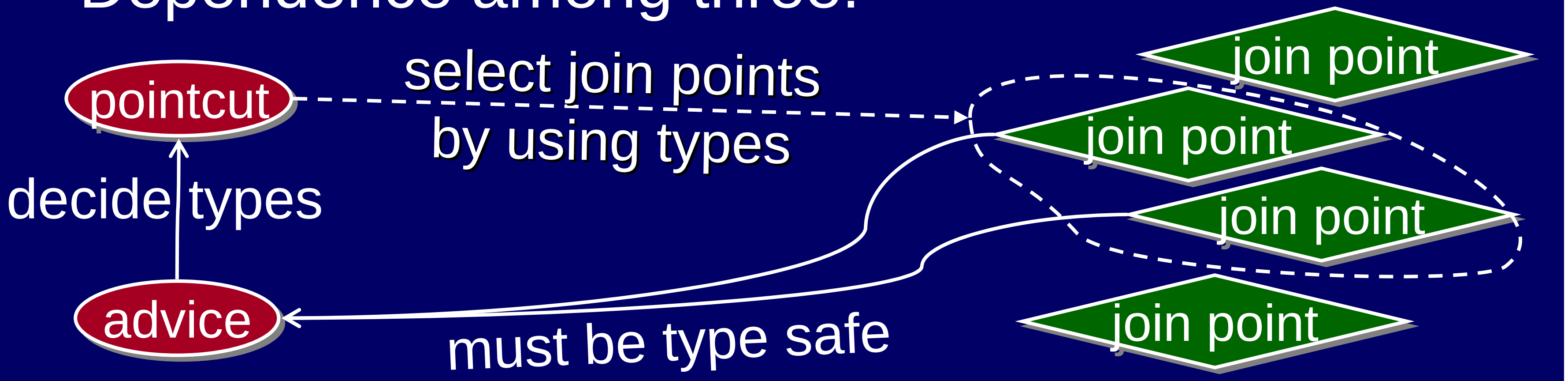
# Type inference in pointcut and advice

- FP: do type inference in pointcuts & advice
- AO: use pointcut types to select join points

```
pointcut evaluation env exp =
  call eval env; (exp:term)
```

appl. to “eval” of type $\alpha \to \text{term} \to \beta$

- Dependence among three:



# Type inference in pointcut and advice: before selection approach

Design decision in Aspectual Caml:

- perform type inference in pointcut & advice ***before selecting join points***

- Advantages:
  - type checking of advice decls. without base program
  - filtering anonymous functions by types
    e.g., “appl to functions of type $\alpha \to$ term $\to \beta$”

# Type inference in pointcut and advice: implementation

1. Infer types in advice and pointcut with
   - type variables to join point values
   - the global type environment
2. Select join points that have more specific types

```
pointcut evaluation env exp = call eval env; exp
advice tr_results env exp = [around evaluation env exp]
   (let v = proceed exp in print_int v; v) : int
```

this is just a name

α→β→int

α

β

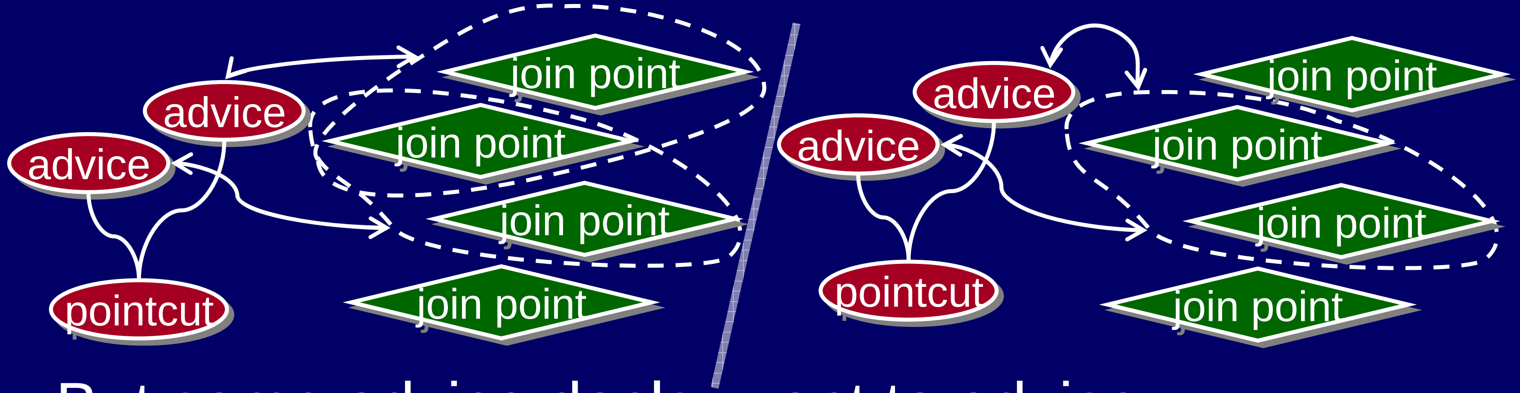
β→int

int→unit

...(eval:env→term→int) env t1...
...(eval:(α → β)→int→int) f 2...
...(eval:int→int→string) 2 3...

# Polymorphic / monomorphic pointcuts

- Advice declarations can share one pointcut
- Type inference ***refines*** types in pointcuts



- But some advice decls. want to advise
  the same set of join points
- Solution: polymorphic / monomorphic pointcuts

# Polymorphic / monomorphic pointcuts: examples

- Polymorphic: (cf. let-polymorphism in ML)

```
pointcut logged x = call _ x && not(within(tracing))
advice trint x = [before logged x]
   print_int x
advice trstr x = [before logged x]
   print_str x
```

any application

only int→α

only str→α

- Monomorphic: prohibits type instantiation ⇒ guarantee to match the same points

```
concrete pointcut logged x =
              call _ (x:int) && not(within(tracing))
advice trbeg x = [before logged x]
   print_str "begins with "; print_int x
advice trend x = [after logged x]
   print_str "ends."
```

trace beginning & end of an application

# Type extension: AO mechanism for data structure

- AspectJ offers two AO mechanisms
  - i.e., pointcut & advice + intertype decls.
  - for crosscutting concerns involving with  
    behavior + data structure  
    e.g., source-level tracing = printing expressions +  
    source code locations in AST nodes
- FP's data structure = variant types
- Approach in Aspectual Caml: ***type extension***
  - extends variant types in two ways

# Type extension: two ways to extend variant types

- Extra fields to a constructor
  - e.g., to associate annotation string to Var

    `type+ term = Var of ... * string {“”}`
  - visible only in the defined aspect

cf. adding fields to a class

default value

- Extra constructors to a type
  - e.g., to define a new syntax to the parse tree

    `type+ term = ... | Trace of term`
  - need to advise all pattern matchings

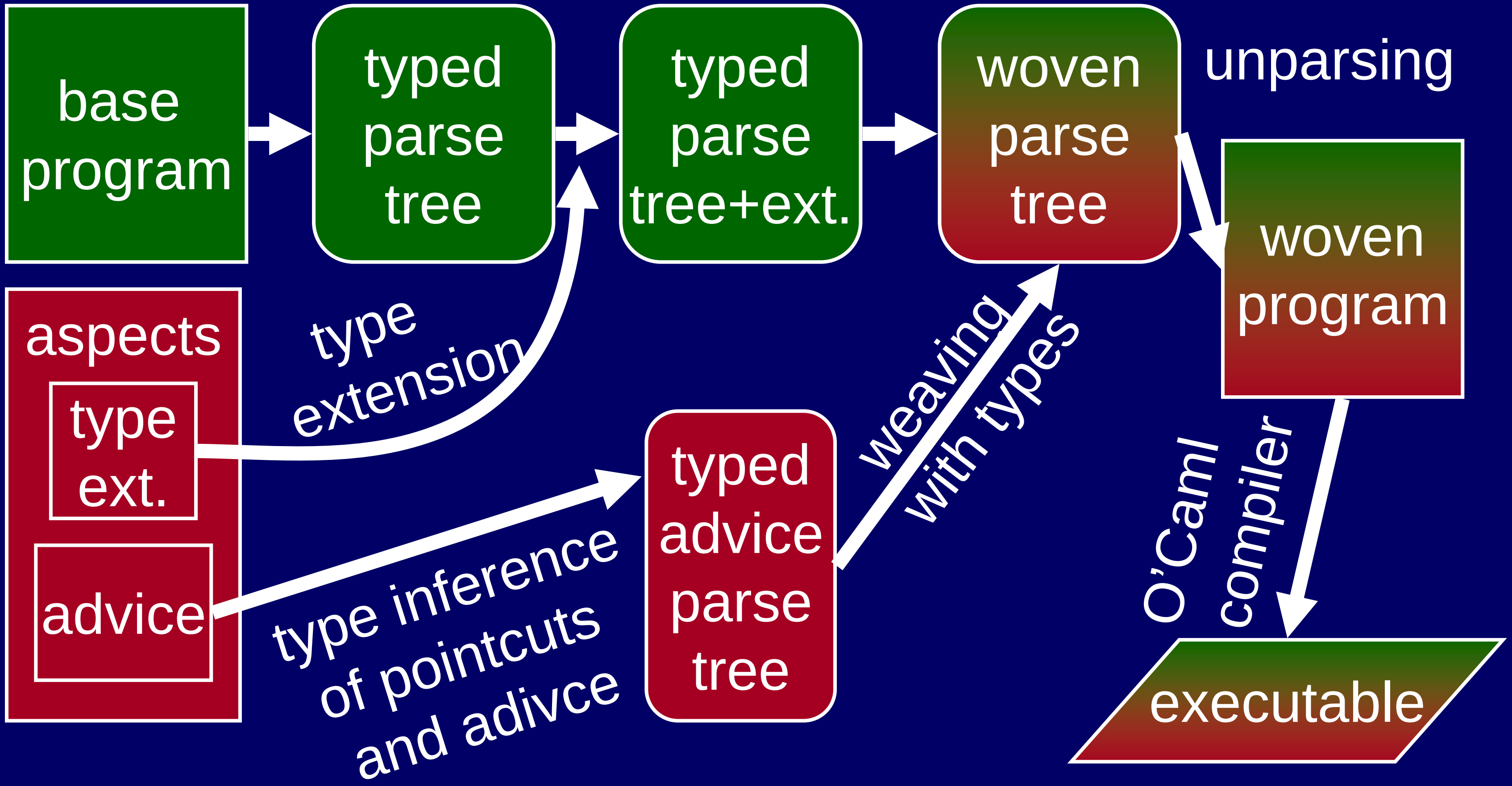
cf. subclassing

# Implementation

- Prototype: a translator to O'Caml
  - – for efficiency of executables
  - – for borrowing backends (e.g., compilers, tools)
- Approach: to modify O'Caml compiler
  - – to extend the syntax
  - – to access type information

  size: 2K LOC addition/modification

# Implementation: compilation process

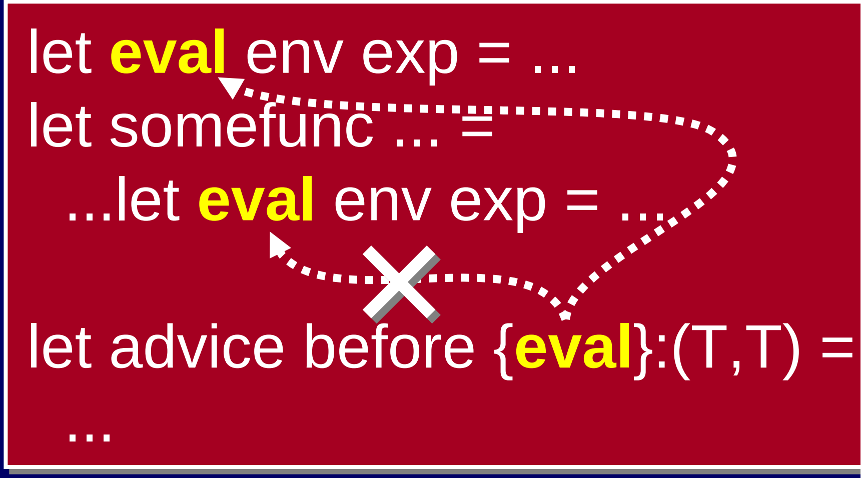


# Related work

- AO typed FPLs: AML[Walker03], PolyAML[Dantas05], TinyAspect[Aldrich03]
  - minimalistic approach (pointcut and advice only)
  - to study type soundness, polymorphism, module systems, etc.
- Extensible data structures
  - polymorphic variants [Garrigue98] + open recursion
  - type safe update programming [Erwig03]

# Related work: PolyAML [Dantas+05]

- Common to Aspectual Caml:
  - polymorphism in advice:
    advice runs at join points of different types
- Different from Aspectual Caml:
  - no polymorphism in pointcuts
  - selecting join points by ***variables*** in current scope
  - static types are used for only checking



```
let eval env exp = ...
let somefunc ... =
  ...let eval env exp = ...

let advice before {eval}:(T,T) =
  ...
```

# Conclusion

- Designed & implemented Aspectual Caml
- Interesting AOP features:
  - Curried pointcuts
  - Type inference before selecting join points
  - Polymorphic / monomorphic pointcuts
  - Type extension
- Future work
  - Formalization, improved implementation, more language features (cf. G’Caml)