PolyAML: A Polymorphic Aspect-oriented Functional Programming Language

Daniel S Dantas
Princeton University

Joint Work with:

David Walker
Geoffrey Washburn
Stephanie Weirich

Princeton University
University of Pennsylvania
University of Pennsylvania
Aspect-oriented Programming

Large Program

Want to add a logging feature that references many parts of the program.
Aspect-oriented Programming

- **When? (Pointcut)**
- **What? (Advice)**

**Added Functionality**
(ILogger)
AOP Advantages

- No Tangling
  - To understand original functionality, examine mainline code
AOP Advantages

- No Scattering
  - To understand added logging functionality, examine logging advice only
AOP Advantages

- Easy Consistency Checking
  - To audit added logging code for consistency, simply analyze pointcut that triggers logging advice
Our Research Area

Our Previous Research: Aspect-oriented Functional Programming
Our Research Area

Our Current Research: Polymorphic Aspect-oriented Functional Programming
Add Polymorphism

- How to write advice that triggers on polymorphic functions?
  - Function can pass many types of data to the advice that it triggers
Add Polymorphism

- How to write advice triggered by many different functions with different types of input?
Contributions

- Design of a Polymorphic Aspect-oriented Functional Programming Language (PolyAML)
- Extension of Damas-Milner type inference algorithm to polymorphic aspects
- Semantics for PolyAML through translation to core language
- Implementation in Standard ML
PolyAML

- Polymorphic First-Class Pointcuts
- Polymorphic Before and After Advice
- Polymorphic Stack Analysis
PolyAML Example

let
rec f(x) = x + 1
rec g(x) = if x then 1 else 0
rec h(x) = 3
...

main code

in
f 1 ─── returns 2
PolyAML Example

let

rec f (x) = x + 1
rec g (x) = if x then 1 else 0
rec h (x) = 3

... val pc = \{ f \} : (int, int) pc

in

f 1 returns 2

\{ f \} : (int, int) pc

main code

pointcut (when)

argument type

result type

in
let

rec f(x) = x + 1
rec g(x) = if x then 1 else 0
rec h(x) = 3

...

val pc = \{f\} : (int, int) pc

advice before pc (arg : int, name) =
    print "before " ^ name
    ^ " " ^ (itos arg); arg

in

f 1 returns 2
PolyAML Example

let

rec f (x) = x + 1
rec g (x) = if x then 1 else 0
rec h (x) = 3
...

val pc = {f} : (int, int) pc
advice before pc (arg : int, name) =
  print "before " ^ name
  ^ " " ^ (itos arg); arg

in

f 1

prints "before f 1" and returns 2
PolyAML Polymorphism

let

rec f(x) = x + 1
rec g(x) = if x then 1 else 0
rec h(x) = 3
...

val pc1 = {f} : (int, int) pc
val pc2 = {f, g} : (?, ?) pc
val pc3 = {f, g, h} : (?, ?) pc
...

advice before pc2 (arg:?, name) =
advice after pc3 (arg:?, name) =

val f : int \rightarrow int
val g : bool \rightarrow int
val h : \forall \alpha. \alpha \rightarrow int

pointcuts (when)
advice (what)
PolyAML Polymorphism

let

rec f(x) = x + 1
rec g(x) = if x then 1 else 0
rec h(x) = 3
...

val pc1 = {f} : (int, int) pc
val pc2 = {f, g} : (? , ?) pc
val pc3 = {f, g, h} : (? , ?) pc
...

advice before pc2 (arg:?, name) =
advice after pc3 (arg:?, name) =

val f : int → int
val g : bool → int
val h : ∀α.α → int

What are the types of these pointcuts?

advice (what)
PolyAML Polymorphism

let
  rec f(x) = x + 1
  rec g(x) = if x then 1 else 0
  rec h(x) = 3
...
  val pc1 = {f} : (int, int) pc
  val pc2 = {f, g} : (?, ?) pc
  val pc3 = {f, g, h} : (?, ?) pc
...
  advice before pc2 (arg:?, name) =
  advice after pc3 (arg:?, name) =

val f : int → int
val g : bool → int
val h : ∀α.α → int

What are the types of these pointcuts?

How does this advice work?
PolyAML Simple Pointcuts

val f : int → int
val g : bool → int
val h : ∀α.α → int

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>
PolyAML Simple Pointcuts

let

rec f(x) = x + 1
rec g(x) = if x then 1 else 0
rec h(x) = 3

val f : int → int
val g : bool → int
val h : ∀α.α → int

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>
PolyAML Simple Pointcuts

let

rec f(x) = x + 1
rec g(x) = if x then 1 else 0
rec h(x) = 3

val f : int → int
val g : bool → int
val h : ∀α.α → int

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>

val fpc = {f} : (int, int) pc
PolyAML Simple Pointcuts

let

\[ \text{rec } f(x) = x + 1 \quad \text{val } f : \text{int} \to \text{int} \]
\[ \text{rec } g(x) = \text{if } x \text{ then } 1 \text{ else } 0 \quad \text{val } g : \text{bool} \to \text{int} \]
\[ \text{rec } h(x) = 3 \quad \text{val } h : \forall \alpha. \alpha \to \text{int} \]

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>g</td>
<td>bool</td>
<td>int</td>
</tr>
</tbody>
</table>

\[ \text{val } fpc = \{ f \} : (\text{int, int}) \text{ pc} \]
PolyAML Simple Pointcuts

let

    rec f(x) = x + 1
    rec g(x) = if x then 1 else 0
    rec h(x) = 3

val f : int → int
val g : bool → int
val h : ∀α.α → int

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>g</td>
<td>bool</td>
<td>int</td>
</tr>
</tbody>
</table>

val fpc = {f} : (int, int) pc
val gpc = {g} : (bool, int) pc
let

rec f(x) = x + 1
rec g(x) = if x then 1 else 0
rec h(x) = 3

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>g</td>
<td>bool</td>
<td>int</td>
</tr>
<tr>
<td>h</td>
<td>α</td>
<td>int</td>
</tr>
</tbody>
</table>

val fpc = {f} : (int, int) pc
val gpc = {g} : (bool, int) pc
let

\[
\begin{align*}
\text{rec } f(x) &= x + 1 \\
\text{rec } g(x) &= \text{if } x \text{ then } 1 \text{ else } 0 \\
\text{rec } h(x) &= 3
\end{align*}
\]

val \( f : \text{int} \to \text{int} \)
val \( g : \text{bool} \to \text{int} \)
val \( h : \forall \alpha. \alpha \to \text{int} \)

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>( g )</td>
<td>bool</td>
<td>int</td>
</tr>
<tr>
<td>( h )</td>
<td>( \alpha )</td>
<td>int</td>
</tr>
</tbody>
</table>

val \( fpc = \{ f \} : (\text{int, int}) \text{ pc} \)
val \( gpc = \{ g \} : (\text{bool, int}) \text{ pc} \)
val \( hpc = \{ h \} : (\forall \alpha. \alpha, \text{int}) \text{ pc} \)
PolyAML Complex Pointcuts

let

\[ \text{rec } f (x) = x + 1 \]
\[ \text{rec } g (x) = \text{if } x \text{ then } 1 \text{ else } 0 \]
\[ \text{rec } h (x) = 3 \]

\[ \text{val } f : \text{int } \rightarrow \text{int} \]
\[ \text{val } g : \text{bool } \rightarrow \text{int} \]
\[ \text{val } h : \forall \alpha. \alpha \rightarrow \text{int} \]

<table>
<thead>
<tr>
<th>Pointcut</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>{f}</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>{g}</td>
<td>bool</td>
<td>int</td>
</tr>
<tr>
<td>{h}</td>
<td>\alpha</td>
<td>int</td>
</tr>
<tr>
<td>{f, g}</td>
<td>\text{int, bool } \rightarrow \alpha</td>
<td>\text{int, int } \rightarrow \text{int}</td>
</tr>
<tr>
<td>{f, h}</td>
<td>\text{int, bool } \rightarrow \alpha</td>
<td>\text{int, int, int } \rightarrow \text{int}</td>
</tr>
</tbody>
</table>
PolyAML Complex Pointcuts

let

\[ \text{rec } f(x) = x + 1 \]
\[ \text{rec } g(x) = \text{if } x \text{ then } 1 \text{ else } 0 \]
\[ \text{rec } h(x) = 3 \]

<table>
<thead>
<tr>
<th>Pointcut</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>{f}</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>{g}</td>
<td>bool</td>
<td>int</td>
</tr>
<tr>
<td>{h}</td>
<td>(\alpha)</td>
<td>int</td>
</tr>
<tr>
<td>{f, g}</td>
<td>{int, bool} (\Rightarrow \alpha)</td>
<td>{int, int} (\Rightarrow) int</td>
</tr>
</tbody>
</table>
PolyAML Complex Pointcuts

let

\[\text{rec } f \ (x) = x + 1\]
\[\text{rec } g \ (x) = \text{if } x \text{ then } 1 \text{ else } 0\]
\[\text{rec } h \ (x) = 3\]

\[\text{val } f : \text{int} \rightarrow \text{int}\]
\[\text{val } g : \text{bool} \rightarrow \text{int}\]
\[\text{val } h : \forall \alpha. \alpha \rightarrow \text{int}\]

<table>
<thead>
<tr>
<th>Pointcut</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>{f}</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>{g}</td>
<td>bool</td>
<td>int</td>
</tr>
<tr>
<td>{h}</td>
<td>(\alpha)</td>
<td>int</td>
</tr>
<tr>
<td>{f, g}</td>
<td>{\text{int, bool}} \Rightarrow \alpha</td>
<td>{\text{int, int}} \Rightarrow \text{int}</td>
</tr>
</tbody>
</table>

\[\text{val } pc2 = \{f, g\} : (\forall \alpha. \alpha, \text{int}) \text{ pc}\]
PolyAML Complex Pointcuts

let

\[\text{rec } f \,(x) = x + 1\]
\[\text{rec } g \,(x) = \text{if } x \text{ then } 1 \text{ else } 0\]
\[\text{rec } h \,(x) = 3\]

\[\text{val } f : \text{int} \to \text{int}\]
\[\text{val } g : \text{bool} \to \text{int}\]
\[\text{val } h : \forall \alpha.\alpha \to \text{int}\]

<table>
<thead>
<tr>
<th>Pointcut</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>{f}</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>{g}</td>
<td>bool</td>
<td>int</td>
</tr>
<tr>
<td>{h}</td>
<td>(\alpha)</td>
<td>int</td>
</tr>
<tr>
<td>{f, g}</td>
<td>{\text{int, bool}} \Rightarrow \alpha</td>
<td>{\text{int, int}} \Rightarrow \text{int}</td>
</tr>
<tr>
<td>{f, g, h}</td>
<td>{\text{int, bool, }\alpha} \Rightarrow \alpha</td>
<td>{\text{int, int, int}} \Rightarrow \text{int}</td>
</tr>
</tbody>
</table>

\[\text{val } pc2 = \{f, g\} : (\forall \alpha.\alpha, \text{int}) \text{ pc}\]
PolyAML Complex Pointcuts

let

\begin{align*}
\text{rec } f \,(x) &= x + 1 \\
\text{rec } g \,(x) &= \text{if } x \text{ then } 1 \text{ else } 0 \\
\text{rec } h \,(x) &= 3
\end{align*}

val \,f : \text{int} \rightarrow \text{int} \\
val \,g : \text{bool} \rightarrow \text{int} \\
val \,h : \forall \alpha. \alpha \rightarrow \text{int}

<table>
<thead>
<tr>
<th>Pointcut</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>{f}</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>{g}</td>
<td>bool</td>
<td>int</td>
</tr>
<tr>
<td>{h}</td>
<td>\alpha</td>
<td>int</td>
</tr>
<tr>
<td>{f, g}</td>
<td>{\text{int, bool}} \Rightarrow \alpha</td>
<td>{\text{int, int}} \Rightarrow \text{int}</td>
</tr>
<tr>
<td>{f, g, h}</td>
<td>{\text{int, bool, \alpha}} \Rightarrow \alpha</td>
<td>{\text{int, int, int}} \Rightarrow \text{int}</td>
</tr>
</tbody>
</table>

val \,pc2 = \{f, g\} : (\forall \alpha. \alpha, \text{int}) \,pc
val \,pc3 = \{f, g, h\} : (\forall \alpha. \alpha, \text{int}) \,pc
PolyAML Complex Pointcuts

let

\[ \text{rec } f \ (x) = x + 1 \]
\[ \text{rec } g \ (x) = \text{if } x \text{ then } 1 \text{ else } 0 \]
\[ \text{rec } h \ (x) = 3 \]

val \( f : \text{int} \rightarrow \text{int} \)
val \( g : \text{bool} \rightarrow \text{int} \)
val \( h : \forall \alpha. \alpha \rightarrow \text{int} \)

<table>
<thead>
<tr>
<th>Pointcut</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {f} )</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>( {g} )</td>
<td>bool</td>
<td>int</td>
</tr>
<tr>
<td>( {h} )</td>
<td>( \alpha )</td>
<td>int</td>
</tr>
<tr>
<td>( {f, \ g} )</td>
<td>{int, \ bool} \Rightarrow \alpha</td>
<td>{int, \ int} \Rightarrow \text{int}</td>
</tr>
<tr>
<td>( {f, \ g, \ h} )</td>
<td>{int, \ bool, \ \alpha} \Rightarrow \alpha</td>
<td>{int, \ int, \ int} \Rightarrow \text{int}</td>
</tr>
</tbody>
</table>

\[ \text{val } pc2 = \{f, \ g\} : (\forall \alpha. \alpha, \text{int}) \ pc \]
\[ \text{val } pc3 = \{f, \ g, \ h\} : (\forall \alpha. \alpha, \text{int}) \ pc \]
\[ \text{val } pc4 = \text{any} : (\forall \alpha. \alpha, \forall \alpha. \alpha) \ pc \]

\[ \text{advice with “any” pointcuts must be prepared to receive any type of data (“most polymorphic”)} \]
PolyAML Pointcuts
PolyAML Pointcuts

- $pc = \{\text{joinpoints}\} : (\forall \alpha. \tau_{\text{arg}}, \forall \beta. \tau_{\text{res}}) pc$
PolyAML Pointcuts

- \( pc = \{\text{joinpoints}\} : (\forall \alpha. \tau_{\text{arg}}, \forall \beta. \tau_{\text{res}}) pc \)

- pointcut must be prepared to receive any type of data that joinpoints might send it

- pointcut is “at least as polymorphic” as its joinpoints
PolyAML Pointcuts

- $pc = \{\text{joinpoints}\} : (\forall \bar{a}. \tau_{\text{arg}}, \forall \bar{\beta}. \tau_{\text{res}}) \Rightarrow pc$

- pointcut must be prepared to receive any type of data that joinpoints might send it

- pointcut is “at least as polymorphic” as its joinpoints

- $\forall \bar{\beta}. \tau_{\text{pointcut}} \leq \forall \bar{a}. \tau_{\text{joinpoint}}$ for all joinpoints
PolyAML Pointcuts

- $pc = \{\text{joinpoints}\} : (\forall \bar{\alpha}. \tau_{\text{arg}}, \forall \bar{\beta}. \tau_{\text{res}}) pc$

- Pointcut must be prepared to receive any type of data that joinpoints might send it.

- Pointcut is “at least as polymorphic” as its joinpoints.

- $\forall \bar{\beta}. \tau_{\text{pointcut}} \leq \forall \bar{\alpha}. \tau_{\text{joinpoint}}$ for all joinpoints

- $\exists \tau. \tau_{\text{pointcut}} [\bar{\tau} / \bar{\beta}] = \tau_{\text{joinpoint}}$ for all joinpoints
PolyAML Pointcuts

- \( pc = \{\text{joinpoints}\} : (\forall \bar{\alpha}. \tau_{\text{arg}}, \forall \bar{\beta}. \tau_{\text{res}}) pc \)

- pointcut must be prepared to receive any type of data that joinpoints might send it

- pointcut is “at least as polymorphic” as its joinpoints

- \( \forall \bar{\beta}. \tau_{\text{pointcut}} \leq \forall \bar{\alpha}. \tau_{\text{joinpoint}} \) for all joinpoints

- \( \exists \tau. \tau_{\text{pointcut}} [\tau/\bar{\beta}] = \tau_{\text{joinpoint}} \) for all joinpoints

\[
\begin{array}{c}
\{f, g\} & \{\text{int, bool}\} \Rightarrow \alpha \\
\forall \alpha \in \text{int} & \forall \alpha \in \text{bool} \\
\forall \alpha [\text{int}/\bar{\alpha}] = \text{int} & \forall \alpha [\text{bool}/\bar{\alpha}] = \text{bool} \\
\end{array}
\]
PolyAML Pointcuts

- \( pc = \{\text{joinpoints}\} : (\forall \alpha. \tau_{\text{arg}}, \forall \beta. \tau_{\text{res}}) \ pc \)

- pointcut must be prepared to receive any type of data that joinpoints might send it

- pointcut is “at least as polymorphic” as its joinpoints

- \( \forall \beta. \tau_{\text{pointcut}} \leq \forall \alpha. \tau_{\text{joinpoint}} \) for all joinpoints

- \( \exists \tau. \tau_{\text{pointcut}} [\bar{\tau} / \bar{\beta}] = \tau_{\text{joinpoint}} \) for all joinpoints

<table>
<thead>
<tr>
<th>{f, g}</th>
<th>{\text{int, bool}} \Rightarrow \alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall \alpha. \alpha \leq \text{int}</td>
<td>\forall \alpha. \alpha \leq \text{bool}</td>
</tr>
</tbody>
</table>
PolyAML Pointcuts

- $\text{pc} = \{\text{joinpoints}\} : (\forall \alpha. \tau_{\text{arg}}, \forall \beta. \tau_{\text{res}}) \text{ pc}$

- pointcut must be prepared to receive any type of data that joinpoints might send it

- pointcut is “at least as polymorphic” as its joinpoints

- $\forall \beta. \tau_{\text{pointcut}} \leq \forall \alpha. \tau_{\text{joinpoint}}$ for all joinpoints

- $\exists \tau. \tau_{\text{pointcut}} [\bar{\tau} / \bar{\beta}] = \tau_{\text{joinpoint}}$ for all joinpoints

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>${f, g}$</td>
<td>${\text{int, bool}} \Rightarrow \alpha$</td>
</tr>
<tr>
<td>$\forall \alpha. \alpha \leq \text{int}$</td>
<td>$\forall \alpha. \alpha \leq \text{bool}$</td>
</tr>
<tr>
<td>$\alpha [\text{int} / \alpha] = \text{int}$</td>
<td>$\alpha [\text{bool} / \alpha] = \text{bool}$</td>
</tr>
</tbody>
</table>
PolyAML Polymorphic Advice
PolyAML Polymorphic Advice

- advice time pointcut \((x : \tau_{\text{pointcut}}, \text{name}) = e_{\text{adv}}\)
  - If pointcut has component type \(\forall\alpha. \tau_{\text{pointcut}}\)
  - then \(x\) is passed value of type \(\tau_{\text{pointcut}}\)
  - and \(e_{\text{adv}}\) must have type \(\tau_{\text{pointcut}}\) for all \(\alpha\)
PolyAML Polymorphic Advice

- advice time pointcut \((x : \tau_{\text{pointcut}}, \text{name}) = e_{\text{adv}}\)
  - If pointcut has component type \(\forall \overline{\alpha}. \tau_{\text{pointcut}}\)
  - then \(x\) is passed value of type \(\tau_{\text{pointcut}}\)
  - and \(e_{\text{adv}}\) must have type \(\tau_{\text{pointcut}}\) for all \(\overline{\alpha}\)

- Remember pointcut is “at least as polymorphic” than its component joinpoints
- So advice is prepared to accept all types of data the joinpoints might send it
PolyAML Polymorphic Advice

val f : int → int
val g : bool → int
val h : ∀α.α → int
let

    rec f (x) = x + 1
    rec g (x) = if x then 1 else 0
    rec h (x) = 3

    ...

    val allpc = {f, g, h} : (\alpha.\alpha, int) pc

in

advice before allpc (arg , name) =
PolyAML Polymorphic Advice

let

rec f (x) = x + 1
rec g (x) = if x then 1 else 0
rec h (x) = 3

... 
val allpc = {f, g, h} : (\forall \alpha. \alpha, int) pc
in

advice before allpc (arg , name) = argument type \alpha
PolyAML Polymorphic Advice

let

rec f (x) = x + 1

rec g (x) = if x then 1 else 0

rec h (x) = 3

...

val allpc = {f, g, h} : (∀α.α, int) pc

in

advice before allpc (arg : α, name) =
print “before ” ^ name; arg

argument type α
let

rec f (x) = x + 1

rec g (x) = if x then 1 else 0

rec h (x) = 3

val allpc = {f, g, h} : (∀α.α, int) pc

in

advice before allpc (arg : α, name) =
    print “before ” ^ name; arg

advice after allpc (res , name) =
PolyAML Polymorphic Advice

let

rec f (x) = x + 1
rec g (x) = if x then 1 else 0
rec h (x) = 3

val f : int → int
val g : bool → int
val h : ∀α.α → int

val allpc = {f, g, h} : (∀α.α, int) pc

in

advice before allpc (arg : α, name) =
  print "before " ^ name; arg

advice after allpc (res , name) =

argument type
  α

result type
  int
PolyAML Polymorphic Advice

let

rec f (x) = x + 1
rec g (x) = if x then 1 else 0
rec h (x) = 3

... val allpc = {f, g, h} : (∀α.α, int) pc

in

advice before allpc (arg : α, name) =
  print “before ” ^ name; arg

advice after allpc (res : int, name) =
  print “after ” ^ name ^ (itos res); res
Type Analysis in Advice

let

rec f (x) =  x + 1
rec g (x) = if x then 1 else 0
rec h (x) = 3

...
val allpc = {f, g, h} : (\alpha.\alpha, int) pc

in

advice before allpc (arg : \alpha, name) =

typcase [unit] \alpha (int => print (itos arg)
  | bool => if arg then print "true" else print "false"
  | _ => ()); arg

val f : int \to int
val g : bool \to int
val h : \forall \alpha.\alpha \to int
Type Analysis in Advice

```ml
let
  rec f (x) = x + 1
  rec g (x) = if x then 1 else 0
  rec h (x) = 3

  ...

  val allpc = {f, g, h} : (\alpha.\alpha, int) pc
```

```ml
in
  advice before allpc (arg : \alpha, name) =
  typecase [unit] \alpha (int => print (itos arg)
    | bool => if arg then print "true" else print "false"
    | _ => ()); arg
```

- Useful enough to get its own macro
- case-advice time pc (x : \tau, name) = e_{adv}
  - Only triggers when joinpoint has type \tau
Polymorphic Case-Advice

val f : int → int
val g : bool → int
val h : ∀α.α → int
let

    rec f (x) = x + 1
    rec g (x) = if x then 1 else 0
    rec h (x) = 3

    ...

    val allpc = \{f, g, h\} : (\forall a.a, int) pc

    ...

    case-advice before allpc (arg : int) =
        print "Int:" ^ (itos arg); arg
Polymorphic Case-Advice

let

\[ \text{rec } f(x) = x + 1 \]
\[ \text{rec } g(x) = \text{if } x \text{ then } 1 \text{ else } 0 \]
\[ \text{rec } h(x) = 3 \]

\[ \ldots \]
\[ \text{val } \text{allpc} = \{ f, g, h \} : (\forall \alpha. \alpha, \text{int}) \text{ pc} \]

\[ \ldots \]
\[ \text{case-advice before allpc } (\text{arg} : \text{int}) = \]
\[ \text{print } \text{"Int:" } ^\wedge (\text{itos } \text{arg}); \text{ arg} \]
\[ \text{case-advice before allpc } (\text{arg} : \text{bool}) = \]
\[ \text{print } \text{"Bool:" } ^\wedge (\text{if } \text{arg then } \text{"true" } \text{ else } \text{"false"}); \text{ arg} \]
let
  rec f (x) =  x + 1
  rec g (x) = if x then 1 else 0
  rec h (x) = 3
...
val allpc = {f, g, h} : (∀α.α, int) pc
...
case-advice before allpc (arg : int) =
  print "Int:" ^ (itos arg); arg
case-advice before allpc (arg : bool) =
  print "Bool:" ^ (if arg then "true" else "false"); arg
in
  f 3;
Polymorphic Case-Advice

let

rec f (x) =  x + 1
rec g (x) = if x then 1 else 0
rec h (x) = 3

...
val allpc = {f, g, h} : (∀α.α, int) pc

...

case-advice before allpc (arg : int) =
print “Int:” ^ (itos arg); arg

case-advice before allpc (arg : bool) =
print “Bool:” ^ (if arg then “true” else “false”); arg

in

f 3;  Prints “Int: 3”
Polyomorphically Case-Advice

let
 rec f (x) = x + 1
 rec g (x) = if x then 1 else 0
 rec h (x) = 3
...
 val allpc = {f, g, h} : (\alpha . \alpha , int) pc
...

case-advice before allpc (arg : int) =
    print “Int:” ^ (itos arg); arg

case-advice before allpc (arg : bool) =
    print “Bool:” ^ (if arg then “true” else “false”); arg

in
 f 3;  ___________ Prints “Int: 3”
g true;
Polymorphic Case-Advice

let

    rec f (x) = x + 1
    rec g (x) = if x then 1 else 0
    rec h (x) = 3

    ...
    val allpc = {f, g, h} : (∀α.α, int) pc
    ...

    case-advice before allpc (arg : int) =
    print “Int:” ^ (itos arg); arg
    case-advice before allpc (arg : bool) =
    print “Bool:” ^ (if arg then “true” else “false”); arg

in

    f 3;  Prints “Int: 3”
    g true;  Prints “Bool: true”
let
  rec f (x) =  x + 1
  rec g (x) = if x then 1 else 0
  rec h (x) = 3

... 
val allpc = {f, g, h} : (∀α.α, int) pc

... 
case-advice before allpc (arg : int) =
  print “Int:” ^ (itos arg); arg

case-advice before allpc (arg : bool) =
  print “Bool:” ^ (if arg then “true” else “false”); arg

in
  f 3;  Prints “Int: 3”
  g true;  Prints “Bool: true”
  h 3;
Polymorphic Case-Advice

let
rec f (x) = x + 1
rec g (x) = if x then 1 else 0
rec h (x) = 3

val f : int → int
val g : bool → int
val h : ∀α.α → int

... val allpc = {f, g, h} : (∀α.α, int) pc

... case-advice before allpc (arg : int) =
    print “Int:” ^ (itos arg); arg
case-advice before allpc (arg : bool) =
    print “Bool:” ^ (if arg then “true” else “false”); arg

in
f 3; Prints “Int: 3”
g true; Displays "Bool: true"
h 3; Displays "False"
Polymorphic Case-Advice

let

rec f (x) = x + 1
rec g (x) = if x then 1 else 0
rec h (x) = 3

... val allpc = {f, g, h} : (\alpha. \alpha, int) pc

... case-advice before allpc (arg : int) =
  print "Int:" ^ (itos arg); arg

  case-advice before allpc (arg : bool) =
    print "Bool:" ^ (if arg then "true" else "false"); arg

  in

  f 3;
  g true;
  h 3;
  h true

Prints "Int: 3"

Prints "Bool: true"
Polymorphic Case-Advice

let
    rec f (x) = x + 1
    rec g (x) = if x then 1 else 0
    rec h (x) = 3

    ...
    val allpc = {f, g, h} : (\alpha.\alpha, int) pc

    ...
    case-advice before allpc (arg : int) =
        print "Int:" ^ (itos arg); arg
    case-advice before allpc (arg : bool) =
        print "Bool:" ^ (if arg then "true" else "false"); arg

in
    f 3;  Prints "Int: 3"
    g true;
    h 3;   Prints "Bool: true"
    h true
PolyAML $\rightarrow$ Core Language

- Provide type-preserving translation to lower "core" language
  - [Walker, Zdancewic, Ligatti '03]

- Provide semantics for "core language"

- Prove properties about "core language"
Core Language

- Polymorphic explicitly-labeled joinpoints
- Polymorphic primitive advice
- Polymorphic stack analysis
Core Labels & Joinpoints

- All PolyAML function calls are joinpoints

- label 1 has type \((\forall \bar{a}. \tau_{\text{lab}})\) label
  - can mark joinpoints of type \(\tau_{\text{lab}}\) for all \(\bar{a}\)

- \(1[\bar{\tau}_{\text{inst}}][[e]]\) creates an explicit joinpoint with label 1
  - type of joinpoint is \(\tau_{\text{lab}}[\bar{\tau}_{\text{inst}}/\bar{a}]\)
Core Labels & Joinpoints

- All PolyAML function calls are joinpoints
- label 1 has type \((\forall \bar{a}. \tau_{lab})\) label
  - can mark joinpoints of type \(\tau_{lab}\) for all \(\bar{a}\)

- \(1[\bar{\tau}_{inst}][[e]]\) creates an explicit joinpoint with
  label 1
  type of joinpoint is \(\tau_{lab} [\bar{\tau}_{inst} / \bar{a}]\)

\[1[\bar{\tau}_{inst}][[v]]\]
Core Labels & Joinpoints

- All PolyAML function calls are joinpoints
- Label $I$ has type $(\forall \bar{a}. \tau_{lab})$ label
  - can mark joinpoints of type $\tau_{lab}$ for all $\bar{a}$
- $I[\bar{\tau}_{inst}][[e]]$ creates an explicit joinpoint with label $I$
  - type of joinpoint is $\tau_{lab}[\bar{\tau}_{inst} / \bar{a}]$

$v : \tau_{lab}[\bar{\tau}_{inst} / \bar{a}]$

advice
Core Labels & Joinpoints

- All PolyAML function calls are joinpoints
- Label $l$ has type $(\forall \alpha. \tau_{lab})$ label
  - can mark joinpoints of type $\tau_{lab}$ for all $\alpha$
- $l[\overline{\tau}_{inst}][[e]]$ creates an explicit joinpoint with label $l$
  - type of joinpoint is $\tau_{lab}[\overline{\tau}_{inst} / \overline{\alpha}]$

$v : \tau_{lab}[\overline{\tau}_{inst} / \overline{\alpha}]$

result : $\tau_{lab}[\overline{\tau}_{inst} / \overline{\alpha}]$

advice
Core Labels & Joinpoints

- All PolyAML function calls are joinpoints
- Label $l$ has type $(\forall \alpha. \tau_{lab})$ label
  - can mark joinpoints of type $\tau_{lab}$ for all $\alpha$
- $l[\overline{\tau}_{inst}][[e]]$ creates an explicit joinpoint with label $l$
  - type of joinpoint is $\tau_{lab}[\overline{\tau}_{inst}/\overline{\alpha}]$

\[ v : \tau_{lab}[\overline{\tau}_{inst}/\overline{\alpha}] \]
\[ \text{result} : \tau_{lab}[\overline{\tau}_{inst}/\overline{\alpha}] \]

result

advice
Core Label Tree

- To translate PolyAML “any” pointcuts, need tree of core labels
- \( l_{\text{child}} = \text{new } (\forall \bar{\alpha}. \tau_{\text{child}}) \leq l_{\text{parent}} \)
  - marks joinpoints of type \( \tau_{\text{child}} \) for all \( \bar{\alpha} \)
  - \( l_{\text{parent}} \) “at least as polymorphic” as new \( l_{\text{child}} \)
Core Label Tree

- To translate PolyAML “any” pointcuts, need tree of core labels
  - $l_{\text{child}} = \text{new} (\forall \bar{\alpha}. \tau_{\text{child}}) \leq l_{\text{parent}}$
    - marks joinpoints of type $\tau_{\text{child}}$ for all $\bar{\alpha}$
    - $l_{\text{parent}}$ “at least as polymorphic” as new $l_{\text{child}}$

$U : \forall \alpha . \alpha$ — most polymorphic label
Core Label Tree

- To translate PolyAML “any” pointcuts, need tree of core labels
  - \( l_{\text{child}} = \text{new} (\forall \bar{\alpha}. \tau_{\text{child}}) \leq l_{\text{parent}} \)
    - marks joinpoints of type \( \tau_{\text{child}} \) for all \( \bar{\alpha} \)
    - \( l_{\text{parent}} \) “at least as polymorphic” as new \( l_{\text{child}} \)

\[ U : \forall \alpha . \alpha \quad \text{most polymorphic label} \]

\[ l_1 : \forall \alpha . \alpha \rightarrow \text{int} \]
Core Label Tree

- To translate PolyAML “any” pointcuts, need tree of core labels
- $l_{child} = \text{new } (\forall \bar{\alpha}. \tau_{child}) \leq l_{parent}$
  - marks joinpoints of type $\tau_{child}$ for all $\bar{\alpha}$
  - $l_{parent}$ “at least as polymorphic” as new $l_{child}$

$U : \forall \alpha . \alpha$ — most polymorphic label

$l_1 : \forall \alpha . \alpha \rightarrow \text{int}$

$l_2 : \text{int}$
Core Label Tree

- To translate PolyAML “any” pointcuts, need tree of core labels

- $l_{\text{child}} = \text{new } (\forall \bar{\alpha}. \tau_{\text{child}}) \leq l_{\text{parent}}$
  - marks joinpoints of type $\tau_{\text{child}}$ for all $\bar{\alpha}$
  - $l_{\text{parent}}$ “at least as polymorphic” as new $l_{\text{child}}$

```
U : \forall \alpha . \alpha
```

```
l_1 : \forall \alpha . \alpha \rightarrow \text{int}
l_2 : \text{int}
l_3 : \text{int} \rightarrow \text{int}
```

most polymorphic label
Core Label Tree

To translate PolyAML “any” pointcuts, need tree of core labels

\( l_{\text{child}} = \text{new} (\forall \bar{\alpha}. \tau_{\text{child}}) \leq l_{\text{parent}} \)
- marks joinpoints of type \( \tau_{\text{child}} \) for all \( \bar{\alpha} \)
- \( l_{\text{parent}} \) “at least as polymorphic” as new \( l_{\text{child}} \)

\[
\begin{align*}
U : \forall \alpha. \alpha & \quad \text{most polymorphic label} \\
 & \quad \downarrow \\
 & l_1 : \forall \alpha. \alpha \to \text{int} \\
 & l_2 : \text{int} \\
 & l_3 : \text{int} \to \text{int}
\end{align*}
\]

- \( l_3 \) joinpoint will trigger \( \{ l_1 \} \) pointcut
Core Label Tree

- To translate PolyAML “any” pointcuts, need tree of core labels
- $l_{\text{child}} = \text{new } (\forall \bar{a}. \tau_{\text{child}}) \leq l_{\text{parent}}$
  - marks joinpoints of type $\tau_{\text{child}}$ for all $\bar{a}$
  - $l_{\text{parent}}$ “at least as polymorphic” as new $l_{\text{child}}$

```
U : \forall a . a
```

```
U : \forall a . a

l_1 : \forall a . a \rightarrow \text{int}

l_2 : \text{int}
```

```
l_3 : \text{int} \rightarrow \text{int}
```

- $l_3$ joinpoint will trigger \{l_1\} pointcut
- all joinpoints will trigger $U$ pointcut
Interpreter

- 5k lines of Standard ML
Interpreter

- 5k lines of Standard ML
**Interpreter**

- 5k lines of Standard ML
Interpreter

- 5k lines of Standard ML

PolyAML → Type Inference → Fully-annotated PolyAML → Translator → Type-safe Core

Type-safe Core → Core Typechecker (verifies translation)
Interpreter

- 5k lines of Standard ML

Diagram:

- PolyAML
  - Type Inference
  - Fully-annotated PolyAML
  - Translator
  - Type-safe Core
  - Core Evaluator
    - Type-safe Core
    - Core Typechecker (verifies translation)

Result
Read the Paper for:

- PolyAML Stack Analysis
- Type Inference for Polymorphic Aspects
- Detailed Core Language Semantics
Related Work

- Aspects and Modules
  - Aspectual Collaborations (Lieberherr, et al. ’03)
  - Open Modules (Aldrich ’04)

- Aspects and Functional Programming
  - Aspect-oriented Scheme (Tucker, et al. ’03)
Related Work
Related Work

- Aspectual Caml (Tatsuzawa, et al)
Related Work

- Aspectual Caml (Tatsuzawa, et al)
  - more varieties of join points
    (wildcards, curried functions, anonymous functions)
Related Work

- Aspectual Caml (Tatsuzawa, et al)
  - more varieties of join points
    - (wildcards, curried functions, anonymous functions)
  - datatypes can be modified by aspects
Related Work

- Aspectual Caml (Tatsuzawa, et al)
  - more varieties of join points (wildcards, curried functions, anonymous functions)
  - datatypes can be modified by aspects

- no formal specification of language semantics
Related Work

- Aspectual Caml (Tatsuzawa, et al)
  - more varieties of join points
    (wildcards, curried functions, anonymous functions)
  - datatypes can be modified by aspects

- no formal specification of language semantics
- no first class pointcuts
Related Work

- Aspectual Caml (Tatsuzawa, et al)
  - more varieties of join points
    - (wildcards, curried functions, anonymous functions)
  - datatypes can be modified by aspects
  - no formal specification of language semantics
  - no first class pointcuts
  - no stack analysis
Future Work
Future Work

- Polymorphic “Around” Advice
Future Work

- Polymorphic “Around” Advice
- Extending type inference to require fewer annotations
  - Pointcuts
  - Typecase statements
Future Work

- Polymorphic “Around” Advice

- Extending type inference to require fewer annotations
  - Pointcuts
  - Typecase statements

- More variety of pointcuts
  - Subtractive pointcuts
  - Aspectual Caml pointcuts
Conclusion

- Design of a Polymorphic Aspect-oriented Functional Programming Language (PolyAML)
- Extension of Damas-Milner type inference algorithm to polymorphic aspects
- Semantics for PolyAML through translation to core language
- Implementation in Standard ML