A Theory of Aspects

David Walker
Princeton University

with Steve Zdancewic, Jay Ligatti
what is an aspect?

- an aspect is a collection of definitions that specifies
  1. a point cut = a set of control flow points
  2. advice = an (effectful) computation that executes at a point cut
aspects: the good

let

fun foo (x) = ...
fun bar (x) = ...
fun baz (x) = ...

... val pc = {foo,bar,baz}

before pc (arg) = print "begin"; arg

after pc (res) = print "end"; res

in ...
aspects: the good

let

fun foo (x) = ...
fun bar (x) = ...
fun baz (x) = ...
...
val pc = {foo, bar, baz}

before pc (arg) = print "begin"; arg

after pc (res) = print "end"; res

in ...

ordinary function declarations
point cut = set of function call sites
aspects: the good

let

fun foo (x) = ...
fun bar (x) = ...
fun baz (x) = ...
...
val pc = {foo, bar, baz}

before pc (arg) = print "begin"; arg

after pc (res) = print "end"; res

in ...

ordinary function declarations
point cut = set of function call sites

advice that runs before/after the calls
modularity via aspects

- aspects: a new way to modularize code
  - aspects specify *when* in the same place as *what*
  - aspects might be used when
    - similar functionality is required at many cfps
    - there is little interaction with the mainline computation
    - the set of cfps is highly susceptible to change
- eg: debugging, profiling, access control code can be centralized in one place
aspects: the bad

let
  fun add1 (x) = x + 1
  fun add2 (x) = add1 (add1 x)

in
  add2 7 (* result = 9 *)
end
aspects: the bad

let
  fun add1 (x) = x + 1
  fun add2 (x) = add1 (add1 x)

  before {add1} (x) = x + 1
in
  add2 7 (* result = 11 *)
end
aspects: the ugly

class adder extends object {
    private fun add1 (x) = x + 1

    public fun add2 (x) = add1 (add1 x)
}

class evil extends object {
    before {add1} (x) = x + 1
}

class victim extends object {
    public fun add4 (x) = add2 (add2 x)
}
aspects: the bad & ugly

- aspects are frightening because:
  1. they break every (?) modularity principle ever invented
  2. implementation has preceded formalization, leaving aspect-oriented programming languages without semantic foundations
what should we do about aspects?

1. forget about them and hope they go away.
   - why pursue ideas that might not work out?

or

2. use PL principles to improve AOPL
   - understand what they are & how they work
   - preserve the good
   - control the bad and the ugly.
this paper

- a strongly typed core calculus of aspects
  - defines two fundamental abstractions:
    - labeled control-flow points
    - first-class advice

- Mini Aspect ML: an idealized AOPL
  - a semantics via typed translation into the core calculus
  - lexical scoping = primitive protection for local invariants
  - a prototype implementation
the rest of the talk

- a core calculus of aspects
- Mini Aspect ML
- wrap-up
the core calculus

- **labeled control flow points:**
  - `new p:t.e` (introduce new label `p` with scope `e`; `p` labels points with type `t`)
  - `e1 <e2>` (use label `e1` by marking the point after execution of `e2`)

- **example:**

  - `new p:int. 3 + p<7 - 2>` \(\rightarrow\) Good
  - `new p:int. 3 + p<true>` \(\rightarrow\) Ill-typed
  - `(new p:int. 3) + p<5>`
the core calculus

- **advice:**
  - \{e_1.x \mapsto e_2\} (introduce advice triggered by e_1; x \mapsto e_2 transforms the data at e_1)
  - e_1 >> e_2 (use advice: run e_1 after all other advice)
  - e_1 << e_2 (use advice: run e_1 before all other advice)
  - eg:

    ```plaintext
    new p:int.
    \{p.x \mapsto x+1\} >> \{p.x \mapsto x*20\} >> p<0>
    evaluates to 20
    ```

    ```plaintext
    new p:int.
    \{p.x \mapsto x+1\} >> \{p.x \mapsto x*20\} << p<0>
    evaluates to 1
    ```
the core calculus

- reminder: labels can only be used within the appropriate scope
  - lexical scoping can be used to protect local invariants from aspect interference
  - eg:

```plaintext
let y = (new p:int. p<0>) in
{p.x => x+1} >>
{p.x => x*20} >>
 y
```

Ill-typed
the core calculus

- the core calculus is powerful:
  - recursion:
    ```
    new p:int.
    \{p.x \Rightarrow \text{let } y = e \text{ in } p\langle x \rangle \} \triangleright>
    p\langle 0 \rangle
    ```
  - references:
    ```
    new ref:int.
    \{ref.x \Rightarrow 0\} \triangleleft
    let get = \lambda \_. \text{ref}\langle 0 \rangle \text{ in}
    let set = \lambda y. \{\text{ref.}_\_ \Rightarrow y\} \triangleright (\) \text{ in}
    (get, set)
    ```

- infinite loop computes e on each iteration
- allocate new location
- run aspects on dummy arg to retrieve stored value
- new aspect run last returns stored value
the core calculus

- advice before & after a function:
  - consider $\lambda x. e : \text{int} \rightarrow \text{bool}$

new before:int.
new after:bool.
{before.y => e1} <<
{after.y => e2} <<
$\lambda x. \text{after<}$

let x = before<x> in
e
>

labels for points before and after
advice before and after
the core calculus

- one more feature:
  - return e1 to e2  \(\text{return e1 to enclosing point e2}\)
  - eg: around advice replacing body of \(\lambda x. e\) with \(e'\):

\[
\begin{align*}
\text{new} & \quad \text{before: int.} \\
\text{new} & \quad \text{after: bool.} \\
\{\text{before}.y =\Rightarrow & \quad \text{return e’ to after}\} \quad << \\
\lambda x. & \quad \text{after}< \\
\text{let} & \quad x = \text{before}<x> \quad \text{in} \\
& \quad e \\
& \quad >
\end{align*}
\]
the rest of the talk

- a core calculus of aspects
- Mini Aspect ML
- wrap-up
Mini Aspect ML (MinAML)

- an idealized external language:
  - types \( t ::= \text{int} \mid t_1 \rightarrow t_2 \)
  - terms \( e ::= \ldots \mid \text{let ds in e} \)
  - dec’s \( ds ::= . \mid \text{(fun f (x:t1):t2 = e) ds} \mid \text{a ds} \)
  - advice \( a ::= \)
    - before \( f(x) = e \)
    - after \( f(x) = e \)
    - around \( f(x) = e \)
    - around \( f(x) = (e_1; \text{proceed y => e2}) \)
Mini Aspect ML (MinAML)

- a type-theoretic interpretation [Harper, Stone]
  - the semantics of MinAML is given by translation into the core calculus:
    \[ \Gamma \vdash e : t \Rightarrow e' \]
    \[ \text{MinAML term} \quad \text{core calculus term} \]
  - the translation is type-preserving
our type-theoretic interpretation of MinAML suggests an implementation strategy:

- MinAML source
  - type theoretic interpretation
  - SML target
    - linking
      - complete program
        - core calculus as SML library
scaling it up

- our framework is robust under extension
- read the paper (!) for:
  - interoperation with first-order Abadi-Cardelli objects
  - point cuts as label sets:
    - let \( pc = \{p1, p2, p3\} \) in \( \{pc.x \Rightarrow e\} \)
  - context-sensitive point cuts
    - regular expressions can be used to match the current run-time stack
the rest of the talk

- a core calculus of aspects
- Mini Aspect ML
- wrap-up
related work

- semantics of aspects
  - denotational semantics for aspects [WKD 2002]
  - typed aspects & aspect combinators [BLW 2002]
  - untyped FP & aspects [TK 2003]
  - untyped OO aspect calculus [JJR 2003]
  - others

- labeled control-flow points
  - Scheme algebraic stepper [CFF 2001]
future work

- from MinAML to Aspect ML proper
  - aspects, modules and information hiding

- reasoning about aspects
  - a theory of program equivalence?

- aspects & security
  - Polymer: a language for composing security policies
conclusions

aspect oriented programming is all the rave & it is not going away

what can we do?

- understand what aspects are and how they work
- use the principles of programming languages to make them safer to use
- embrace relativism