On the Pursuit of Static and Coherent Weaving

WANG Meng (speaker), National University of Singapore

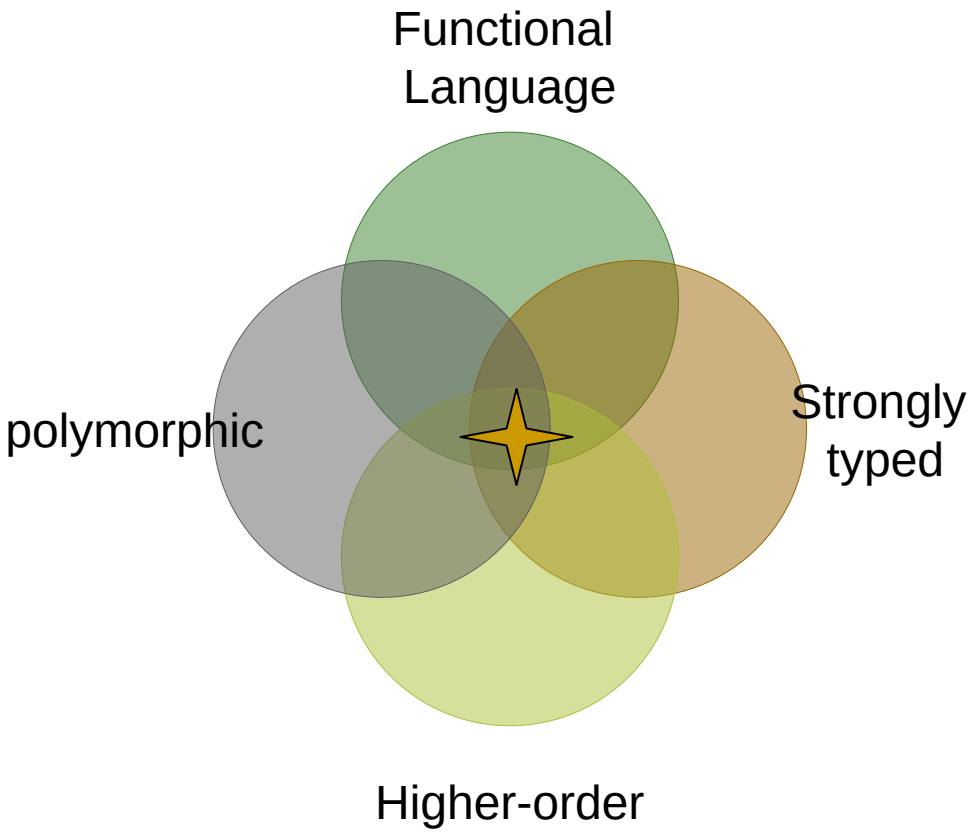
CHEN Kung, National Chengchi University, Taiwan

KHOO Siau-Cheng, National University of Singapore

AOP Languages

- AOP based on Object Oriented languages
  - Java (AspectJ,Jboss,Aspectwerkz,etc)
  - C++ (AspectC++)
- AOP based on Functional languages
  - OCaml (Aspectual Caml)
  - SML (AspectML)

Our Language



Main Mechanisms of Aspects

- ***Introduction*** (injecting new members into existing classes)
- ***Advising*** (transforming computations by intercepting events).

Today's Topic

- ***Introduction*** (injecting new members into existing classes)
- ***Advising*** (transforming computations by intercepting events).
  - *Execution* pointcuts
  - *Around* advices

Weaving

- Translating into a “less-aspect-oriented” intermediate language
  - **Static** – making as many weaving decisions at compilation time as possible
  - **Coherent** – giving the same set of advices to different invocations of a function with inputs of the same type

Weaving -- Challenges

```
n1@advice around {h} (arg::Int)
    = proceed (arg+1) in
n2@advice around {h} (arg) = arg in
let h x = x in
let f x = h x in
(f 1)+(h 2)
```

$Int \rightarrow Int$

$\forall a.a \rightarrow a$

$\forall a.a \rightarrow a$

$\forall a.a \rightarrow a$

Weaving -- Challenges

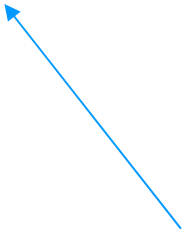
```
n1@advice around {h} (arg::Int)
    = proceed (arg+1) in
n2@advice around {h} (arg) = arg in
let h x = x in
let f x = h x in
(f 1)+(h 2)
```

$Int \rightarrow Int$

$\forall a.a \rightarrow a$

$\forall a.a \rightarrow a$

$\forall a.a \rightarrow a$



n1 and n2

Weaving -- Challenges

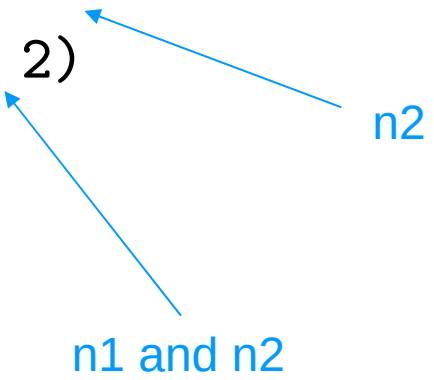
```
n1@advice around {h} (arg::Int)
    = proceed (arg+1) in
n2@advice around {h} (arg) = arg in
let h x = x in
let f x = h x in
(f 1)+(h 2)
```

$Int \rightarrow Int$

$\forall a.a \rightarrow a$

$\forall a.a \rightarrow a$

$\forall a.a \rightarrow a$



Weaving – Advised Types

- ***Advised type*** $f : \forall \overline{a}.(h : t_1).t_2$

  The execution of any application of `f` *may require advices of* `h` *applied with type which should be no more general than* $\forall \overline{a}.t_1$.

Weaving – Advised Types

- ***Advised type*** $f : \forall \overline{a}.(h : t_1).t_2$

  The execution of any application of `f` *may require advices of* `h` *applied with type which should be no more general than* $\forall \overline{a}.t_1$.

```
n1@advice around {h} (arg::Int)
    = proceed (arg+1) in
n2@advice around {h} (arg) = arg in
let h x = x in
let f x = h x in
(f 1)+(h 2)
```

$Int \rightarrow Int$

$\forall a.a \rightarrow a$

$\forall a.a \rightarrow a$

$\forall a.(h : a \rightarrow a).a \rightarrow a$

Weaving – Advised Types

- **Type Directed Translation**

$$\Gamma \vdash_{\rightsquigarrow} e : \sigma \rightsquigarrow e'$$

Weaving – Translation

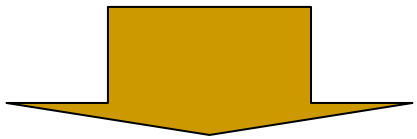
```
n1@advice around {h} (arg::Int)
    = proceed (arg+1) in
n2@advice around {h} (arg) = arg in
let h x = x in
let f x = h x in
(f 1)+(h 2)
```

$Int \to Int$

$\forall a.a \to a$

$\forall a.a \to a$

$\forall a.(h : a \to a).a \to a$



```
let n1 = \arg -> proceed (arg+1) in
let n2 = \arg -> proceed arg in
let h x = x in
let f dh x = dh x in
(f <h,{n1,n2}> 1) + (<h,{n1,n2}> 2)
```

Intermediate Language

- Expressions and values are extended

$$v ::= \ldots \mid \langle v, \{\overline{v}\}\rangle$$
$$e ::= \ldots \mid \langle e, \{\overline{e}\}\rangle$$

- The reduction rules

$$\begin{array}{lcl}
(\lambda x.e\ v) & \longmapsto_\beta & (e[v/x]) \\
(\mathsf{let}\ x = v\ \mathsf{in}\ e) & \longmapsto_\beta & (e[v/x]) \\
(\langle v, \{\}\rangle\ v') & \longmapsto_\beta & (v\ v') \\
(\langle v, \{v_1, \overline{v}\}\rangle\ v') & \longmapsto_\beta & (v_1[\langle v, \{\overline{v}\}\rangle/\mathit{proceed}]\ v')
\end{array}$$

Contributions

Static and Coherent weaving of

- recursive function definitions
- advising other advices' bodies
- higher-order advices

Recursive Functions -- Challenges

```
let g x = x + 1 in
n@advice around {f} (arg:[Int])
  = Cons (g (head arg)) (proceed arg) in
let f x = if (length x) > 0 then f (tail x) else x
in f [1,2,3]
```

$$f : \forall a.(f : [a] \rightarrow [a]).[a] \rightarrow [a]$$

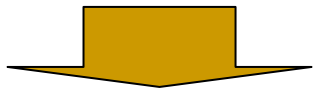
Recursive Functions -- Challenges

```
let g x = x + 1 in
n@advice around {f} (arg:[Int])
  = Cons (g (head arg)) (proceed arg) in
let f df x = if (length x) > 0 then df (tail x) else x
in f ? [1,2,3]
```

$$f : \forall a.(f : [a] \rightarrow [a]).[a] \rightarrow [a]$$

Recursive Functions -- Translation

```
let g x = x + 1 in
n@advice around {f} (arg:[Int])
  = Cons (g (head arg)) (proceed arg) in
let f x = if (length x) > 0 then f (tail x) else x
in f [1,2,3]
```



```
let g x = x + 1 in
let n = \arg.(Cons (g (head arg)) (proceed arg)) in
let f df x = if (length x) > 0
             then df (tail x) else x in
(let F = \y.<f y,{n}> F in F) [1,2,3]
```

Advising Advice Bodies -- Motivation

- Aspects are not limited to observing base programs. Inside the bodies of advice definitions, there may be calls to other functions that are advised. We call these *nested* advices.

Nested Advices -- Example

```
let discount item = (getRate item) * (getPrice item) in
let calcPrice cart = sum (map discount cart) in
calcPrice [1,2,3]
```

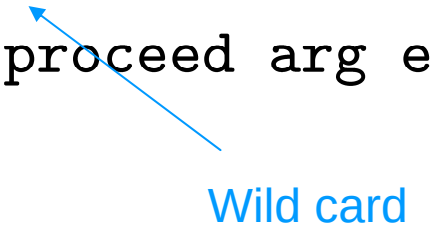
Nested Advices -- Example

```
let discount item = (getRate item) * (getPrice item) in
let calcPrice cart = sum (map discount cart) in
calcPrice [1,2,3]
```

```
n1@advice around {getRate} (arg) =
   (getHolidayRate arg) * (proceed arg)
n2@advice around {getRate} (arg) =
   (getAnniversaryRate arg) * (proceed arg)
```

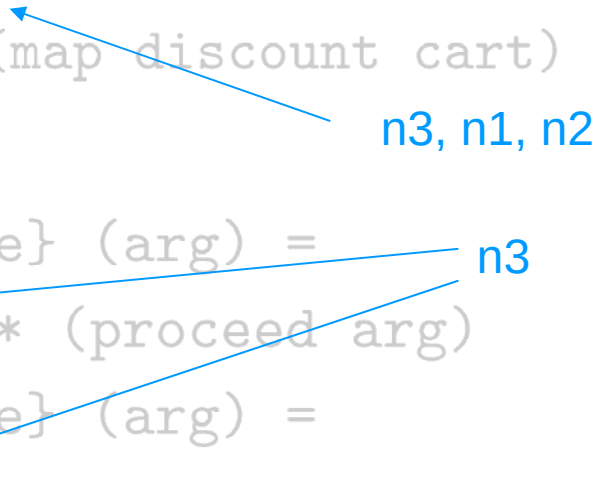
Nested Advices -- Example

```
let discount item = (getRate item) * (getPrice item) in
let calcPrice cart = sum (map discount cart) in
calcPrice [1,2,3]

n1@advice around {getRate} (arg) =
   (getHolidayRate arg) * (proceed arg)
n2@advice around {getRate} (arg) =
   (getAnniversaryRate arg) * (proceed arg)

n3@advice around {get$Rate} (arg:Int) =
   if (arg > 0) then proceed arg else proceed 0
```



Nested Advices -- Example

```
let discount item = (getRate item) * (getPrice item) in
let calcPrice cart = sum (map discount cart) in
calcPrice [1,2,3]

n1@advice around {getRate} (arg) =
   (getHolidayRate arg) * (proceed arg)
n2@advice around {getRate} (arg) =
   (getAnniversaryRate arg) * (proceed arg)
```




```
n3@advice around {get$Rate} (arg:Int) =
   if (arg > 0) then proceed arg else proceed 0
```

Nested Advices – Challenges

- Advice chainings only appear in the woven program which is not a subject for further weaving.
- The typing context where an advice *n* is chained may not be sufficiently specific for another advice to be chained to calls inside *n*'s body.

Higher-Order Advices -- Example

```
let discount item = (getRate item) * (getPrice item) in
let calcPrice cart = sum (map discount cart) in
calcPrice [1,2,3]

n1@advice around {getRate} (arg) =
   (getHolidayRate arg) * (proceed arg)
n2@advice around {getRate} (arg) =
   (getAnniversaryRate arg) * (proceed arg)

n4@advice around {n1,n2} (arg) =
   let finalRate = proceed arg
   in  if (finalRate < 0.5) then 0.5
       else finalRate
```

Nested Advices -- Translation

```
let discount item = (getRate item) * (getPrice item) in
let calcPrice cart = sum (map discount cart) in
calcPrice [1,2,3]

n1@advice around {getRate} (arg) =
   (getHolidayRate arg) * (proceed arg)
...
n3@advice around {get$Rate} (arg:Int) =
   if (arg > 0) then proceed arg else proceed 0
```

$n1 : \forall a.(getHolidayRate : a \rightarrow Real).a \rightarrow Real$

$discount : \forall a.(getRate : a \rightarrow Real).a \rightarrow Real$

Nested Advices -- Translation

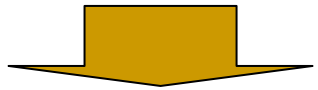
```
let discount item = (getRate item) * (getPrice item) in
let calcPrice cart = sum (map discount cart) in
calcPrice [1,2,3]
```

```
n1@advice around {getRate} (arg) =
   (getHolidayRate arg) * (proceed arg)
...
n3@advice around {get$Rate} (arg:Int) =
   if (arg > 0) then proceed arg else proceed 0
```

$$\text{(Var-A)}\quad \frac{x : \sigma_x \in \Gamma \quad \bar{\sigma} \not\trianglelefteq \sigma_x \quad \Gamma \vdash_{\rightsquigarrow} n_i : [\![\sigma_x]\!] \rightsquigarrow e_i \quad \bar{n} : \bar{\sigma} \bowtie x \in \Gamma \quad \ldots}{\Gamma \vdash_{\rightsquigarrow} x : \sigma_x \rightsquigarrow \langle x\ , \{e_i\}\rangle}$$

Nested Advices -- Translation

```
n3@advice around {get*Rate} (arg:Int) =
   if (arg > 0) then proceed arg else proceed 0 in
n1@advice around {getRate} (arg) =
   (getHolidayRate arg) * (proceed arg) in
let discount item = (getRate item) * (getPrice item) in
let calcPrice cart = sum (map discount cart) in
calcPrice [1,2,3]
```



```
let n3 arg = if (arg > 0)
             then proceed arg else proceed 0
let n1 dh arg = (dh arg) * (proceed arg) in
let calcPrice dc cart = sum (map dc cart) in
let discount dr item = (dr item) * (getPrice item) in
calcPrice (discount <getRate,{n3,n1 <getHolidayRate,{n3}>}>)
          [1,2,3]
```

Higher-Order Advices -- Translation

```
let discount item = (getRate item) * (getPrice item) in
let calcPrice cart = sum (map discount cart) in
calcPrice [1,2,3]

n1@advice around {getRate} (arg) =
   (getHolidayRate arg) * (proceed arg)
n2@advice around {getRate} (arg) =
   (getAnniversaryRate arg) * (proceed arg)

n4@advice around {n1,n2} (arg) =
   let finalRate = proceed arg
   in  if (finalRate < 0.5) then 0.5
       else finalRate
```

Higher-Order Advices -- Translation

$$\text{(Var-A)}\quad \frac{x : \sigma_x \in \Gamma \quad \bar{\sigma} \not\trianglelefteq \sigma_x \quad \Gamma \vdash_{\rightsquigarrow} n_i : [\![\sigma_x]\!] \rightsquigarrow e_i \quad \bar{n} : \bar{\sigma} \bowtie x \in \Gamma \quad \ldots}{\Gamma \vdash_{\rightsquigarrow} x : \sigma_x \rightsquigarrow \langle x\ , \{e_i\}\rangle}$$

```
let n4 arg = let finalRate = proceed arg
             in  if (finalRate < 0.5) then 0.5
                 else finalRate
let n1 arg = (getHolidayRate arg) * (proceed arg) in
let n2 arg = (getAnniversaryRate arg) * (proceed arg) in
let calcPrice cart = sum (map discount cart) in
let discount item = (<getRate,{<n1,{n4}>,<n2,{n4}>}> item)
                    * (getPrice item) in
calcPrice [1,2,3]
```

Correctness of translation

Theorem 1 (Conservative Extension) *Given a program P consisting of a set of advices and a closed base program e. If*

$$\vdash P : \sigma \rightsquigarrow P',$$

then

$$\vdash e : [\![\sigma]\!].$$

Correctness of translation

Theorem 2 (Type Preservation) *Given a program P consisting of a set of advices and a closed base program. If*

$$\vdash P : \sigma \rightsquigarrow P',$$

then

$$\vdash_i P' : \eta(\sigma).$$

$$\begin{aligned} \eta(\forall \overline{a}.\rho) &= \forall \overline{a}.\eta(\rho) \\ \eta((x:t).\rho) &= t \to \eta(\rho) \\ \eta(t) &= t \end{aligned}$$

Related Works

- PolyAML(ICFP 05) by Dantas, Walker, Washburn and Weirich
  - Polymorphic higher-order language
  - First-class pointcuts
  - Dynamic type checking and label matching
  - Only *before* and *after* advices (extension for *around* on progress)
- Aspectual Caml (ICFP 05) by Masuhara, Tatsuzawa and Yonezawa.
  - Higher-order and currying
  - Static introduction
  - Weaver traverses type annotated expressions to insert advice calls. (syntactical)
- Type-directed weaving (PEPM 06) by Wang, Chen and Khoo
  - Polymorphic higher-order language with type scoped *around* advices
  - Static and coherent weaving
  - No recursive functions, nested advices and higher-order advices

Conclusion

- Static and coherent weaving of aspect-oriented functional programs with recursive functions, nested advices and higher-order advices
- Future work:
  - Control-related *Cflow* pointcuts
  - Separate compilation