Typing for a Minimal Aspect Language

Peter Hui, James Riely
DePaul CTI
{phui,jriely}@cs.depaul.edu
Minimal Aspect Calculus
- First presented: Bruns, Jagadeesan, et. al (CONCUR'04)
  - First version of µABC
  - source/target/message model
  - No types
  - Sketches of encodings into µABC
    - untyped λ-calculus w/aspects (subset of minAML (Walker, Zdancewic, Ligatti))
    - object language
- FOAL ‘06: Temporal variant
- This paper:
  - Nontemporal, polyadic version
  - Provide types for µABC
  - Provide full translations into µABC
    - typed, advised λ-Calculus
    - typed, advised object language
  - Translations type-preserving. i.e.:
    - well-typed λ-Calc term =>
      well-typed µ-term
    - well-typed object term =>
      well-typed µ-term
Example term:

```
new a;
new b;
new c;
adv(b -> call<c>);
adv(a -> call<b>);
call<a>;
```

```role
declarations
```
```
adv
declarations
```
```
current
event
```
new a;
new b;
new c;
adv(b -> call<c>)
adv(a -> call<b>)
call<a>;

new a;
new b;
new c;
adv(b -> call<c>)
adv(a -> call<b>)
[adv(a -> call<b>)]<a>;

'declarations remain constant'

'match' triggers advice lookup

'matching advice (LIFO)'

'current event'
new a;
new b;
new c;
adv(b -> call<c>);
adv(a -> call<b>);
[adv(a -> call<b>)]<a>;
call<b>;

new a;
new b;
new c;
adv(b -> call<c>);
adv(a -> call<b>);
call<b>;
new a;
new b;
new c;
adv(b -> call<c>);
adv(a -> call<b>);
[adv(b -> call<c>)] <b>;

μABC

new a;
new b;
new c;
adv(b -> call<c>);
adv(a -> call<b>);
call<c>;}
“proceed” variable, hierarchical roles:

\[
\begin{align*}
\text{declarations} & \\
\text{new } c; & \\
\text{new } f; & \\
\text{new int; } & \\
\text{new 10:int; } & \\
\text{adv}(f,x:\text{int } \rightarrow \text{call}<c,x>); & \\
\text{adv}(z;f,x:\text{int } \rightarrow z<f,x+1>); & \\
\text{call}<f,10>; &
\end{align*}
\]
new c;
new f;
new int;
new 10:int;
adv(f,x:int -> call<c,x>);
adv(z.f,x:int -> z<f,x+1>);
call<f, 10>;

new c;
new f;
new int;
new 10:int;
adv(f,x:int -> call<c,x>);
adv(z.f,x:int -> z<f,x+1>);[
adv(f,x:int -> call<c,x>);
adv(z.f,x:int -> z<f,x+1>);]
<f, 10>;
new c;
new f;
new int;
new 10:int;
adv(f, x:int -> call<c, x>);
adv(z.f, x:int -> z<f, x+1>);
[adv(f, x:int -> call<c, x>);
adv(z.f, x:int -> z<f, x+1>);]
<f, 10>;

new c;
new f;
new int;
new 10:int;
adv(f, x:int -> call<c, x>);
adv(z.f, x:int -> z<f, x+1>);
[adv(f, x:int -> call<c, x>);]
<f, 10+1>;}
new c;
new f;
new int;
new 10:int;
adv(f,x:int -> call<c,x>);
adv(z.f,x:int -> z<f,x+1>);
[adv(f,x:int -> call<c,x>);]
<f, 10+1>;

new c;
new f;
new int;
new 10:int;
adv(f,x:int -> call<c,x>);
adv(z.f,x:int -> z<f,x+1>);
call<c,10+1>;

Note: We have:
- Obliviousness:
  - advice body localized within advice
  - advice can be added without altering program text
- Quantification
  - pointcuts specify which events trigger advice
Typing

How can a term get stuck?

1. new f;
   adv(z; f -> z<f>)
   call<f>

   new f;
   adv(z; f -> z<f>)
   [adv(z; f -> z<f>)]<f>

   new f;
   adv(z; f -> z<f>)
   <><f>

- Advice proceeds, but with no enqueued advice.
How can a term get stuck?

2. ...
   new f;
   new g;
   adv(z;f,x,c ->call<c,x>)
   adv(z;f,x,c ->z<g,x>)
   call<f,10,k>

   ...
   adv(z;f,x,c ->call<c,x>)
   adv(z;f,x,c->z<g,x>)
   [adv(z;f,x,c ->call<c,x>)]
   <g,10>

   adv(z;f,x,c ->call<c,x>)
   adv(z;f,x,c->z<g,x>)
   [adv(z;f,x,c ->call<c,x>)]
   <f,10,k>

-Advice:
  - proceeds
  - alters event
  - new event no longer compatible with remaining advice
How can a term get stuck?

3. new f:int->int;
new 10:int;
new k:int⁻¹;
call<f,10,k>;

Bad: call returns nothing :-(

Idea:
- Type events
- Type advice
- Ensure all types “agree”

Event Types:
Example:
new int; new 5:int; new f;
adv(f, x:int -> M);
call<f,5>

<f,5> has type <f, int>

Advice Types:
Example:
adv(f, x:int -> M) also has same type
A note on our running example…

Roles:
- int: “Integer”
- int->int: “Function taking an int, returning an int”
- int-1: “Continuation (c.f. CPS) of type int”

```
new int : top;
new int->int : top;
new int⁻¹ : top
new f:int->int;
new 10:int;
new k:int⁻¹;
adv(z;f,x:int,c: int⁻¹ -> z<f,x,c>);
call<f,10,k>;
```
Typing

“Advice proceeds, but with no enqueued advice”

**Solution:** advice “finality” (=doesn’t proceed)

```
new f:int->int;
new 10:int;
new k:int-1;
adv(z; f, x:int, c: int-1 -> z<f,x,c>)
call<f,10,k>;
```

red advice is final;
<f,int, int-1> has been finalized

```
new f:int->int;
new 10:int;
new k:int-1;
adv(z; f, x:int, c: int-1 -> z<f,x,c>)
adv(f, x:int, c: int-1 -> call<c,x>)
adv(z; f, x:int, c: int-1 -> z<f,x,c>)
call<f,10,k>;
```

i.e., this is bad...

...but this is OK.
Typing

red advice has type \( \langle f, \text{int, int}^-1 \rangle \)
(same type as event)

new f:int->int;
new 10:int;
new k:int^-1;
call\(<f, 10, k>\);

Also bad: call returns nothing :-(

...but this is OK;
red advice has
type \( \langle f, x: \text{int, c: int}^-1 \rangle \)
“Advice proceeds, alters event, new event no longer compatible with remaining advice”

**Solution:** Ensure that:
1. Events always agree with enqueued advice
2. Proceeds always agree with enqueued advice

$$\text{adv}(z; f, x: \text{int}, c: \text{int}^1 \rightarrow M, \text{adv}(z; g, g, g, g \rightarrow N))$$

<i.e., this is bad (pointcuts not compatible w/ event, not compatible w/ each other)</i>

$$\text{adv}(z; g, y: \text{int} \rightarrow M, \text{adv}(z; g, x: \text{int} \rightarrow N))$$

<i>Solution: Constraint:
1. pointcuts must agree with each other
2. pointcuts must agree with event.</i>
Typing

\[ \text{adv}(z; f, x: \text{int}, c: \text{int}^1 \to z<3>) \]
\[<f, 39, k>;\]

i.e., this is bad (proceeds to incompatible event)

\[ \text{adv}(z; f, x: \text{int}, c: \text{int}^1 \to z<f>) \]
\[<f, 39, k>;\]

Solution: If it proceeds, must proceed to event of same type.
Typing

If it doesn’t proceed, event type can change...

...but it still must be well typed! e.g.: bad:

\[
\text{[adv}(z; f, x: \text{int, c: int}^1 \rightarrow \text{call}<g>)]
\text{<f,39,k>};
\]

...OK

\[
\text{[adv}(z; f, x: \text{int, c: int}^1 \rightarrow \\
\quad \text{[adv}(g \rightarrow M]<g>)]
\text{<f,39,k>};
\]

\[
\text{[adv}(z; f, x: \text{int, c: int}^1 \rightarrow \\
\quad \text{[adv}(g \rightarrow M]<g>)]
\text{<f,39,k>};
\]
Rules look like this:

As \(<Us>\) “ok” if:
1. All advice in As have same type as Us
2. There is some nonproceeding advice in As
3. All advice in As is well-typed

\(\text{adv}(z; f, x: \text{int}, c: \text{int}^{-1} \rightarrow M)\) well typed if:
1. \(M\) “ok” with \(x: \text{int}, c: \text{int}^{-1}\)

\(\text{call}\langle Us\rangle\) “ok” if exists some advice of same type as Us.
Types

Why distinguish between exact/inexact advice?

Suppose we don’t distinguish:

new f:int->int;  new g:int->int;
adv( g, x:int, y:int⁻¹ -> M);
   // would have type <int->int, int, int⁻¹>
   // therefore, <int->int, int, int⁻¹> finalized.
call<f, 40, k>;
   // would have type <int->int, int, int⁻¹>

Since <int->int, int, int⁻¹> finalized, and event has same type, this is well-typed!
Why distinguish between exact/inexact advice?

Thus we make the distinction:
new f:int->int; new g:int->int;
adv( g, x:int, y:int⁻¹ -> M);
  // has type <g, int, int⁻¹>
call< f, 40, k>;
  // has type <f, int, int⁻¹>
<g, int, int⁻¹> finalized, <f, int, int⁻¹> not. Therefore not well typed.
Types

Why distinguish between exact/inexact advice?

Note: Requires caller, advice to “agree” on “calling protocol”. e.g.: caller must know when to mark roles exact.

Future work: redefine type system to allow for completely oblivious calling convention
Translation: Advised \( \lambda \)-Calculus --> \( \mu \)ABC

\( \lambda \)-Calculus Syntax:

\[
\begin{align*}
A & ::= \lambda x.M \\
D & ::= \text{fun } f=A \mid \text{adv}(z.f\to A) \\
U,V & ::= n \mid \text{unit} \mid A \\
M,N & ::= V \mid UV \mid zU \mid D;M \mid \text{let } x=M;N
\end{align*}
\]
Translation: Advised λ-Calculus -
-> μABC

Example:

fun f = \( \lambda x. x^2 \);

fun f = \( \lambda x. x^2 \);

f(10) 10^2
Translation: Advised λ-Calculus -> μABC

Translation of λ-term with continuation k

```
fun f = \(x \rightarrow x^2\);
f(10)
```

```
fun f = \(x \rightarrow x^2\);
10^2
```

```
new f;
adv(f, x, c -> call<c, x^2>);
call<f, 10, k>
```

```
new f;
adv(f, x, c -> call<c, x^2>);
[adv(f, x, c -> call<c, x^2>)]f, 10, k
```

“Protocol” <function, arg, continuation>
Translation: Advised \( \lambda \)-Calculus -
\( \rightarrow \) \( \mu \text{ABC} \)

\[
\text{fun } f = \lambda x. x^2; \\
f(10)
\]

\[
\text{new } f; \\
\text{adv}(f, x, c \rightarrow \text{call}<c,x^2>); \\
[\text{adv}(f, x, c \rightarrow \text{call}<c,x^2>)]<f,10,k>
\]
Translation: Advised $\lambda$-Calculus -> $\mu$ABC

Example with advice:

```
fun f = $\lambda$x.x^2;
adv (z.f -> $\lambda$y.z(y+1));
f(10)
```

(10+1)^2

(semantiscs c.f. Walker et.al. (minAML))
Translation: Advised $\lambda$-Calculus $\rightarrow \mu$ABC

fun f=$\lambda x.x^2$;
adv (z.f -> $\lambda y.z(y+1)$);
f(10)

new f;
adv(z.f,x,c -> call<c,x^2>);
adv(z.f,y,c -> z(f,y+1,c) );
call<f,10,k>;

fun f=$\lambda x.x^2$;
adv (z.f -> $\lambda y.z(y+1)$);
(\lambda y. (\lambda x.x^2)(y+1)) 10

new f;
adv(z.f,x,c -> call<c,x^2>);
adv(z.f,y,c -> z(f,y+1,c) );
[adv(z.f,x,c -> call<c,x^2>);
adv(z.f,y,c -> z<f,y+1,c>)]<f,10,k>;
Translation: Advised \( \lambda \)-Calculus -

\[
\begin{align*}
\text{fun } f &= \lambda x. x^2; \\
\text{adv } (z.f \rightarrow \lambda y. z(y+1)); \\
(\lambda y. (\lambda x. x^2)(y+1)) &= 10
\end{align*}
\]
Translation: Advised $\lambda$-Calculus - $\rightarrow$ $\mu$ABC

fun $f = \lambda x. x^2$;
adv (z. $f \rightarrow \lambda y. z(y+1)$);
($\lambda x. x^2)(10+1)$

fun $f = \lambda x. x^2$;
adv (z. $f \rightarrow \lambda y. z(y+1)$);
$(10+1)^2$

ew $f$;
adv(z. $f, x, c \rightarrow \text{call}<c, x^2>$);
adv(z. $f, y, c \rightarrow z(f, y+1, c)$);
[adv(z. $f, x, c \rightarrow \text{call}<c, x^2>$);]
<$f, 10+1, k>$;

ew $f$;
adv(z. $f, x, c \rightarrow \text{call}<c, x^2>$);
adv(z. $f, y, c \rightarrow z(f, y+1, c)$);
call $<k, (10+1)^2>$;
Translation: Another Example

fun f = \x.x^2;
fun g = \x.x^3;
adv(z.f -> \y.let v = g(y);
    z(v);)
f(10)

fun f = \x.x^2;
fun g = \x.x^3;
adv(z.f -> \y.let v = g(y);
    z(v);)
let v = g(10);
(\x.x^2) v
Translation: Another Example

fun f = \x. x^2;
fun g = \x. x^3;
adv(z.f -> \y. let v = g(y);
    z(v);)
let v = (\x. x^3) 10;
(\x. x^2) v

(\x. x^2) (10^3)

fun f = \x. x^2;
fun g = \x. x^3;
adv(z.f -> \y. let v = g(y);
    z(v);)
let v = 10^3;
(\x. x^2) v

(10^3)^2
Translation: Advised Object Language --> µABC

Object Language Syntax:

A ::= λx.M
C ::= cls a:b(ls = As);
D,E ::= obj p:a | advc{z;a.l -> A}
M,N ::= v | v.l(us) | z(us) | A(us) | D;M |
       let x=M;N
Translation: Advised Object Language --> µABC

Example:

\[
\text{cls } c(\ l=\lambda x.x^2); \\
\text{obj } o: c; \\
\text{advc}(z;c.l->\lambda y.z(y+1)) \\
o.l(5); \\
(\lambda y.\lambda x.x^2(y+1)) 5 \\
(5+1)^2
\]
Translation: Advised Object Language --> \( \mu ABC \)

```plaintext
cls c( l=lx.x^2);
obj o:c;
advc(z;c.l->\lambda y.z(y+1))
o.l(5);

ew c; new l;
adv(self:c, l,x,d-> call<d,x^2>);
new o:c;
adv(z; self:c,l,y,d-> z<c,l,y+1,d>);
call<o,l,5,k>;

new c; new l;
adv(self:c, l,x,d-> call<d,x^2>);
new o:c;
adv(z; self:c,l,y,d-> z<c,l,y+1,d>); 
[adv(self:c, l,x,d-> call<d,x^2>),
adv(z; self:c,l,y,d-> z<c,l,y+1,d>)]
<o,l,5,k>;

new c; new l;
adv(self:c, l,x,c-> call<c,x^2>);
new o:c;
[adv(self:c, l,x,d-> call<d,x^2>)]

ew c; new l;
adv(self:c, l,x,c-> call<c,x^2>);
new o:c;
call <k,(5+1)^2>
```
Correctness of Translations

Establish “correctness” by showing translation preserved by evaluation:

\[ M_\mu \leftrightarrow N_\mu \]
\[ M_\lambda \leftrightarrow N_\lambda \]

Then \( N_\lambda \sim N'_\lambda \)
Correctness of Translations

`~` defined via “structural congruence”:

- “in certain cases, order is irrelevant”:
  
  
  new f; new g;
  new g; ~ new f;

- “in certain cases, we can hoist stuff out of advice bodies”
  
  adv(f)->{new g; call<x>;} ~ new g;
  ...
  adv(f)->{call<x>;} ...
  ...
Correctness of Translations

Biggest hurdle: advice lookup

fun f = λx.x; 
adv(z.f->λy.z(y+1)); 
f(3);

new f; 
adv(f,x,c->call<c,x>); 
adv(z.f,y,c->z<f,y+1,c>); 
call<f,3,k> 

(λy. (λx.x)(y+1))3 

new g; 
adv(g,y,c->new h;...); 
call <g,3,k> 

~ (!) 

... 
[adv(f,x,c->call<c,x>), 
adv(z.f,y,c->z<f,y+1,c>)] 
<f,3,k>
Future work

- Establish semantic equivalence between µABC terms (e.g., formalize correctness of ‘~’)
- Redefine λ-semantics
  - “slow down” advice substitution in λ to be more like µABC semantics