Typing for a Minimal Aspect Language

Peter Hui, James Riely

DePaul CTI

{phui,jriely}@cs.depaul.edu

μABC

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

µABC

- FOAL '06: Temporal variant
- This paper:
  - Nontemporal, polyadic version
    - Provide types for µABC
    - Provide full translations into µABC
      - typed, advised λ-Calculus
      - typed, adviced object language
  - Translations type-preserving. i.e.:
    - well-typed λ-Calc term => well-typed µ-term
    - well-typed object term => well-typed µ-term

µABC

Example term:

role declarations

new a;

new b;

new c;

adv(**b** -> call<c>)

adv(**a** -> call<b>)

advice declarations

call<a>;

current event

µABC

declarations remain constant

```
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>;
```

‘call’ triggers advice lookup

matching advice (LIFO)

current event

µABC

declarations remain constant

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

→

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

advice evaluation

µABC

new a;

new b;

new c;

adv(**b** ->call<c>)

adv(**a** ->call<b>)

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

→

new a;

new b;

new c;

adv(**b** ->call<c>)

adv(**a** ->call<b>)

call<c>;

µABC

"proceed" variable, hierarchical roles:

declarations

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>;

µABC

declarations remain constant

```
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>;
```

Advice "queue"

Current event

μABC

```
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>;
```

μABC

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.

Typing

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>)*
adv(z;f,x,c->z<g,x>)]
<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>

→ ???

-Advice:
- *proceeds*
- *alters event*
- *new event no longer compatible with remaining advice*

Typing

How can a term get stuck?

3.

new f:int->int;
new 10:int;
new k:int^{-1};
call<**f**,10,k>;

$\not\rightarrow$

Bad: call returns nothing :-(

μABC

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

Typing

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"

<u>Solution</u>: advice "finality" (=doesn't proceed)

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>)
call<f,10,k>;
```

i.e., this is bad…

```
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>;
```

…but this is OK.

Typing

red advice has type <**f**,int, int^{-1}>
(same type as event)

new f:int->int;
new 10:int;
new k:int^{-1};
adv(**f**,x:int,c: int^{-1} -> call<c,x>)
call<**f**,10,k>;

…but this is OK;
red advice has
type <**f**,x:int,c: int^{-1}>

new f:int->int;
new 10:int;
new k:int^{-1};
call<**f**,10,k>;

Also bad: call returns nothing :-(

Typing

"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

[*adv(z;**f**,x:int,c: int^{-1} -> M,*
*adv(z;**g**,**g**,**g**,**g** -> N)*]
<**g**,39>;

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

[*adv(z;**g**,y:int -> M,*
*adv(z;**g**, x:int -> N)*]
<**g**,39>;

Solution: Constraint:

1. pointcuts must agree with each other
2. pointcuts must agree with event.

Typing

[*adv(z;**f**,x:int,c: int*$^{-1}$ *-> z<3>*]
<**f**,39,k>;

[*adv(z;**f**,x:int,c: int*$^{-1}$ *-> z<f>*]
<**f**,39,k>;

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

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

Typing

[*adv(z;**f**,x:int,c:* int^{-1} *-> call<**g**>*]
<**f**,39,k>;

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

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

[*adv(z;**f**,x:* *,c:* int^{-1} *->*
[adv(-> M)]<***g***>**]**
f,39,k>;

[*adv(z;**f**,x:int,c:* int^{-1} *->*
[adv(**g** -> M)]<***g***>**]**
<**f**,39,k>;

…OK

Typing

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

adv(z; f,x:int,c:int^{-1} -> M) well typed if:

1. M "ok" with x:int,c:int^{-1}

call<Us> "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!

Types

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 λ-Calculus --> μABC

λ-Calculus Syntax:

A ::= λx.M

D ::= fun f=A |

adv(z.f->A)

U,V ::= n | unit | A

M,N ::= V | UV | zU | D;M | let x=M;N

Translation: Advised λ-Calculus --> μABC

Example:

fun f=λx.x^2;
f(10)

→

fun f=λx.x^2;
10^2

Translation: Advised λ-Calculus -> μABC

Translation of λ-term with continuation k

```
fun f=λx.x^2;
f(10)
```

→

```
fun f=λx.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 λ-Calculus --> µABC

```
fun f=λx.x^2;
f(10)
```

→

```
fun f=λx.x^2;
10^2
```

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

→

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

Translation: Advised λ-Calculus --> μABC

Example with advice:

fun f=λx.x^2;
adv (z.f -> λy.z(y+1));
f(10)

→

fun f=λx.x^2;
adv (z.f -> λy.z(y+1));
(λy. (λx.x^2)(y+1)) 10

→*

fun f=λx.x^2;
adv (z.f -> λy.z(y+1));
(10+1)^2

(semantics c.f. Walker et.al. (minAML))

Translation: Advised λ-Calculus --> μABC

fun f=λx.x^2;
adv (z.f -> λy.z(y+1));
f(10)

→

fun f=λx.x^2;
adv (z.f -> λy.z(y+1));
(λy. (λx.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));
call<**f**,10,k>;

→

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 λ-Calculus --> µABC

fun f=λx.x^2;
adv (z.f -> λy.z(y+1));
(λy.(λx.x^2)(y+1)) 10

→

fun f=λx.x^2;
adv (z.f -> λy.z(y+1));
(λx.x^2)(10+1)

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>;

→

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>);*]
<**f**,10+1,k>;

Translation: Advised λ-Calculus --> µABC

```
fun f=λx.x^2;
adv (z.f -> λy.z(y+1));
(λx.x^2)(10+1)
```

→

```
fun f=λx.x^2;
adv (z.f -> λy.z(y+1));
(10+1)^2
```

```
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>);]
    <f,10+1,k>;
```

→

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

Translation: Another Example

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

→

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

→

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

→

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

Translation: Another Example

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

→

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

→

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

→

```
fun f=λx.x^2;
fun g=λx.x^3;
adv(z.f -> λy.let v=g(y);
               z(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

<u>Example:</u>

```
cls c( l=λx.x^2);
obj o:c;
advc(z;c.l->λy.z(y+1))
o.l(5);
```

→

```
cls c( l=λx.x^2);
obj o:c;
advc(z;c.l->λy.z(y+1))
(λy.λx.x^2(y+1)) 5
```

→*

```
cls c( l=λx.x^2);
obj o:c;
advc(z;c.l->ly.z(y+1))
(5+1)^2
```

Translation: Advised Object Language --> µABC

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

```
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>;
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>;
```

→

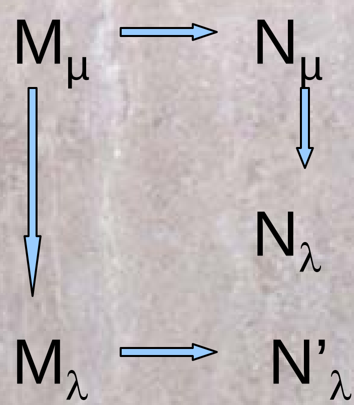
```
new c; new l;
adv(self:c, l,x,c-> call<c,x^2>);
new o:c;
[adv(self:c, l,x,d->
call<d,x^2>)]
```

→

```
new 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:



Then $N_\lambda \sim N'_\lambda$

Correctness of Translations

'~' defined via "structural congruence":

- "in certain cases, order is irrelavent":

```
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 Traslations

Biggest hurdle: advice lookup

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

→ (λy. (λx.x)(y+1))3

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

~ (!)

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

END