

Aspect-oriented C (V 0.5)

Quick Reference

Terminology

ACC

ACC is ASPECT-ORIENTED C implemented by *acc*, the Aspect-oriented C Compiler.

aspect

Aspect encapsulate non-modular system concerns, like security policies, transaction support, synchronization concerns etcetera. ACC represents aspects as C files containing C declarations and statements, and ACC syntax, such as pointcuts and advice.

join point

A *join point* is a well-defined point in the execution context of a program. ACC supports *call* and *execution* join points. A *call* join point is the point where a function is called. An *execution* join point is the point where a function is executed.

pointcut

A *pointcut* is a language extension representing one or more join points. ACC supports *primitive pointcuts*, *composite pointcuts*, and *named pointcuts*.

advice

An *advice* represents the code to be executed when a join point matches a pointcut defined inside the advice declaration. ACC supports the *before*, *after*, and *around* advice.

Pointcut

args(int, int)

The join points of calling and executing functions taking (int, int) as parameter type.

call(void foo(int))

The join points of calling function foo.

callp(void foo(int))

The join points of calling function foo by dereferencing a function pointer.

cflow(call(void foo(int, int)))

The join points under the control flow of calling function foo.

execution(void foo(int))

The join points of executing function foo.

infile("t1.mc")

The join points in the input file "t1.mc".

infunc(foo)

The join points inside the foo's function body.

pointcut MyPC(): call(void foo(int));

A named pointcut MyPC() representing the join points of calling function foo. MyPC() can be used as a pointcut.

result(int)

The join points of calling and executing functions whose return type is int.

call(void foo()) && infunc(main)

The join points of calling function foo inside function main.

call(void foo()) && ! infunc(main)

Calls of function foo, except those called inside main.

call(void foo()) || call(void bar())

Calls of either function foo or function bar.

call(void foo()) && cflow(call(void bar()))

Calls of function foo in the control flow of calling function bar.

general form

args(a list of types or identifiers)
[call|callp|execution](function-signature)
cflow(pointcut)
infile("file name")
infunc(identifier)

pointcut pointcut-name (parameter-list):pointcut;
result(type or identifier)
pointcut-1 && pointcut-2
pointcut-1 || pointcut-2
! pointcut
(pointcut)

Wildcard Matching

call(i\$t f\$oo(in\$))

This represents any call to functions starting with "f" and ending in "oo", having a return type starting with "i" and ending in "t", and accepting one parameter having a type starting with "in," such as "int foo(int)" or "it f2oo(in)".

args(int, ..., char)

This represents any call or execution of functions accepting an int and a char as first and last parameters, such as "void foo(int, char)" or "int foo2(int, char*, char)".

call(int foo(int)) && infunc(foo\$o2)

This represents any call of function "foo" inside functions whose name starts with "fo" and ends in "o2".

general form

"\$": matches any type identifier or any continuous length string, including the empty string.

"...": matches any length item list, including the empty list.

Advice

before():execution(void foo(int)){...}

Advice code runs *before* the execution of function foo.

after():call(void foo(int)){...}

Advice code runs *after* calling function foo.

int around():call(int foo(char)){...}

Advice code runs *instead* of calling function foo.

before (*int a*):*call(void foo (int))* \$\$ *args(a)* {... }

Advice code runs *before* calling function `foo`, and variable “a” holds the parameter value of function `foo` and can be used inside the advice code.

after (*int a*):*call(int foo (void))* \$\$ *result(a)* {... }

Advice code runs *after* calling function `foo`, and variable “a” holds the return value of function `foo` and can be used inside the advice code.

before (*int a, int b*): *cflow(call(void foo(int) && args(b))) && call(int foo2 (int)) && args(a)* {... }

Advice code runs *before* calling function `foo2` in the control flow of calling function `foo`, and variable “a” holds the parameter value of function `foo2` and variable “b” holds the parameter value of function `foo`. Both “a” and “b” can be used inside the advice code.

general form

```
type-specifieropt before|after|around ( parameter-type-listopt ): pointcut  
{ function-body }
```

special identifiers inside advice body

this→*funcName*

A string representation of the function name of the match join point.

this→*kind*

A string representation of the join point kind, either “call” or “execution”.

proceed()

Only used inside `around` advice. It takes the original value of the arguments, and calls or executes the original function.

Examples using special identifiers

```
void around(): call(int foo()) {  
    printf(“%s,%s”,this→kind,this→funcName);  
    proceed() ; }
```

Static Crosscutting

ACC also provides mechanism to support static crosscutting, such as the addition of members to structs and unions.

introduce() : **intype**(*struct X*) {

double b;

int d;

}

A member “double b” and “int d” will be inserted at the *end* of the definition of type “struct X”.

general form

```
intype(type-name)  
introduce (): pointcuts { member-declarations }
```

Example

The following is a reusable tracing aspect.

```
before(): call($ $(..)) && cflow(execution($  
main(..))) {  
    printf(“calling %s\n”,this→funcName);  
}
```

Use ACC Compiler

Suppose the above aspect is saved in file “a.acc”, and the core file (i.e., the file not containing ACC syntax) is save in “b.mc”.

1. Copy files to have .c suffix

```
>cp a.acc a_acc.c
```

```
>cp b.mc b_mc.c
```

2. Preprocess the files by a preprocessor, and save the output in files with the by the ACC compiler required suffixes. This step is necessary because `gcc` does not recognize the .acc and .mc suffix. However, if the preprocessor, like `cpp`, is not picky about the file suffix, this step could be skipped.

```
>gcc -E a_acc.c > a_acc.acc
```

```
>gcc -E b_mc.c > b_mc.mc
```

3. Perform ACC compilation (i.e., weaving)

```
>acc a_acc.acc b_mc.mc
```

4. Perform compilation

```
>gcc a_acc.c b_mc.c
```

command line option

1. -h , -help

Display help information.

2. -t , -thread-safe

The code generated to support the `cflow()` pointcut is thread-safe (based on specific `gcc` functionality).

3. -v , -version

The compiler’s Version number is printed.

4. -af=<suffix> , -aspect-suffix=<suffix>

Specify file suffix for aspect file.

5. -mf=<suffix> , -mainfile-suffix=<suffix>

Specify file suffix for non-aspect file.

For up to date information, please refer to <http://www.AspectC.net>.

©Copyright 2007 Middleware Systems Research Group, University of Toronto, Weigang (Michael) Gong and Hans-Arno Jacobsen. All Rights Reserved.