Aspect-Oriented Programming with AspectC++

Part III – Tool Support

Overview

- ac++ compiler
  - open source and base of the other presented tools
- ag++ wrapper
  - easy to use wrapper around g++ for make-based projects
- AspectC++ plugin for Eclipse®
  - sophisticated environment for AspectC++ development

About ac++

- Available from **www.aspectc.org**
  - Linux, Win32, MacOS binaries + source (GPL)
  - documentation: Compiler Manual, Language Reference, ...
- Transforms AspectC++ to C++ code
  - machine code is created by the back-end (cross-)compiler
  - supports g++ language extensions
- Current version: 2.2
  - front end is based on Clang 3.9.2

Aspect Transformation

```
aspect Transform {
  advice call("% foo()") : before() {
    printf("before foo call\n");
  }
  advice execution("% C::%()") : after()
{
    printf(tjp->signature ());
  }
};
```

Transform.ah

```
class Transform {
  static Transform __instance;
  // ...
  void __a0_before () {
    printf ("before foo call\n");
  }
  template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
      printf (tjp->signature ());
    }
};
```

Transform.ah'

Aspect Transformation

```
aspect Transform {
  advice call("% foo()") : before() {
    printf("before foo call\n");
  }
  advice execution("% C::%()") : after()
{
    printf(tjp->signature ());
  }
};
```

Transform.ah

Aspects are transformed into **ordinary classes**

```
class Transform {
  static Transform __instance;
  // ...
  void __a0_before () {
    printf ("before foo call\n");
  }
  template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
      printf (tjp->signature ());
    }
};
```

Transform.ah'

Aspect Transformation

```
aspect Transform {
  advice call("% foo()") : before() {
    printf("before foo call\n");
  }
  advice execution("% C::%()") : after()
{
    printf(tjp->signature ());
  }
};
```

Transform.ah

One global aspect **instance** is created by default

```
class Transform {
  static Transform __instance;
  // ...
  void __a0_before () {
    printf ("before foo call\n");
  }
  template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
      printf (tjp->signature ());
    }
};
```

Transform.ah'

Aspect Transformation

```
aspect Transform {
  advice call("% foo()") : before() {
    printf("before foo call\n");
  }
  advice execution("% C::%()") : after()
{
    printf(tjp->signature ());
  }
};
```

Transform.ah

Advice becomes a **member function**

```
class Transform {
  static Transform __instance;
  // ...
  void __a0_before () {
    printf ("before foo call\n");
  }
  template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
      printf (tjp->signature ());
    }
};
```

Transform.ah'

Aspect Transformation

```
aspect Transform {
  advice call("% foo()") : before() {
    printf("before foo call\n");
  }
  advice execution("% C::%()") : after()
{
    printf(tjp->signature ());
  }
};
```

Transform.ah

```
class Transform {
  static Transform __instance;
  // ...
  void __a0_before () {
    printf ("before foo call\n");
  }
  template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
      printf (tjp->signature ());
    }
};
```

Transform.ah'

"Generic Advice" becomes a **template member function**

Joinpoint Transformation

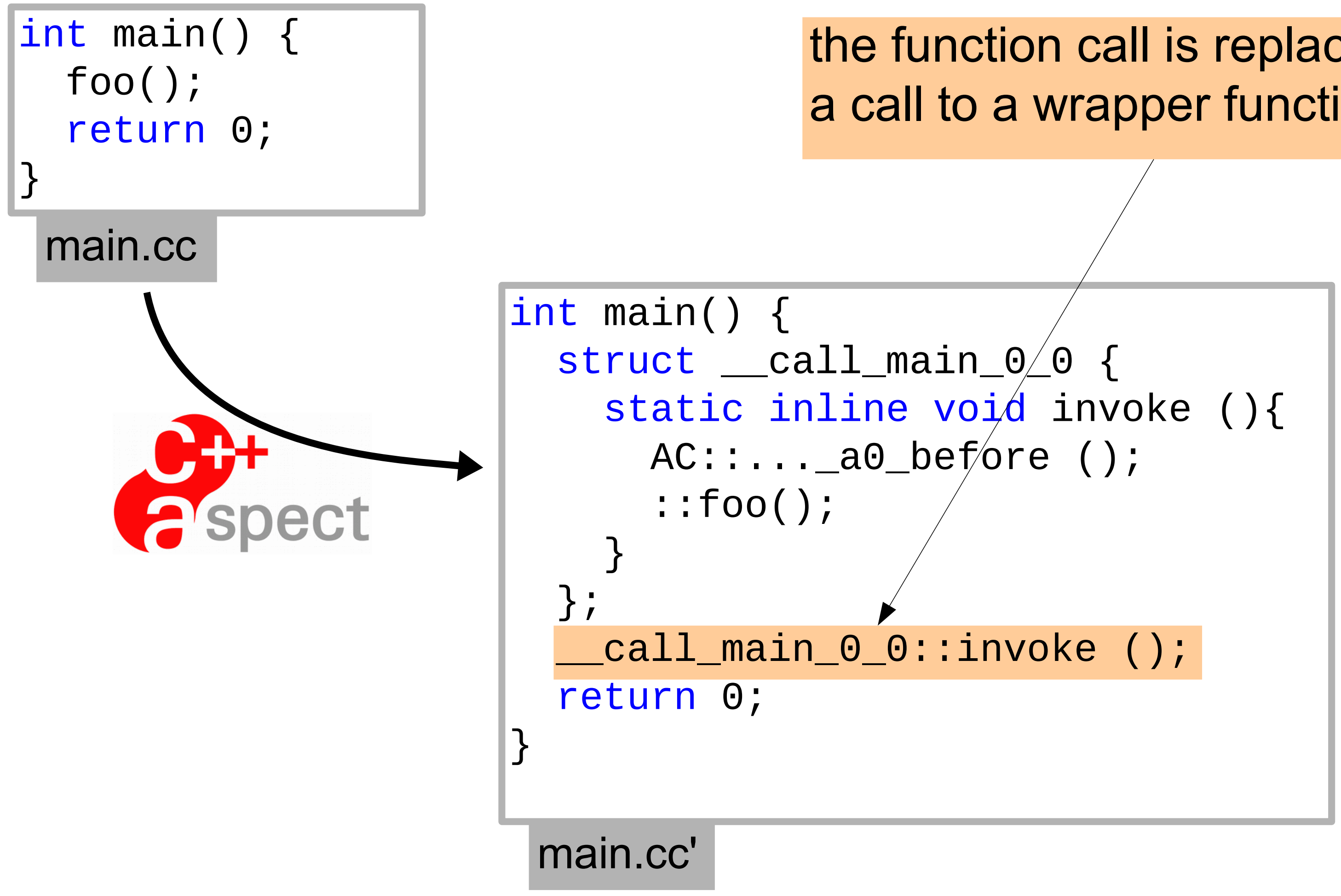
```
int main() {
  foo();
  return 0;
}
```

main.cc

```
int main() {
  struct __call_main_0_0 {
    static inline void invoke (){
      AC::..._a0_before ();
      ::foo();
    }
  };
  __call_main_0_0::invoke ();
  return 0;
}
```

main.cc'

Joinpoint Transformation



```
int main() {
  foo();
  return 0;
}
```

main.cc

the function call is replaced by a call to a wrapper function

```
int main() {
  struct __call_main_0_0 {
    static inline void invoke (){
      AC::..._a0_before ();
      ::foo();
    }
  };
  __call_main_0_0::invoke ();
  return 0;
}
```

main.cc'

Joinpoint Transformation

```
int main() {
  foo();
  return 0;
}
```

main.cc

a local class invokes the advice code for this joinpoint

```
int main() {
  struct __call_main_0_0 {
    static inline void invoke (){
      AC::..._a0_before ();
      ::foo();
    }
  };
  __call_main_0_0::invoke ();
  return 0;
}
```

main.cc'

Translation Modes

- Whole Program Transformation-Mode
  - e.g. `ac++ -p src -d gen -e cpp -Iinc -DDEBUG`
  - transforms whole directory trees
  - generates manipulated headers, e.g. for libraries
  - can be chained with other whole program transformation tools
- Single Translation Unit-Mode
  - e.g. `ac++ -c a.cc -o a-gen.cc -p .`
  - easier integration into build processes

Tool Demo

- AspectC++ plugin for Eclipse®
  - sophisticated environment for AspectC++ development

Summary

- Tool support for AspectC++ programming is based on the ac++ command line compiler
  - full “obliviousness and quantification”
  - delegates the binary code generation to your favorite compiler
- Non-commercial IDE integration is available
  - Eclipse®