Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

Werner Dietl

Software Component Technology

Exercises 8: Extended Typing

Wintersemester 03/04

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

C++ `const`

- "When using a pointer, two objects are involved: the pointer itself and the object pointed to. 'Prefixing' a declaration of a pointer with `const` makes the object, but not the pointer, a constant. To declare a pointer itself, rather than the object pointed to, to be a constant, we use the declarator operator `*const` instead of plain `*`."
- Bjarne Stroustrup: "*The C++ Programming Language, Third Edition*", page 94

C++ `const` is not transitive

```
class Attr {
  Attr(int i) {x = i;}
  int x;
};

class A {
  A() { attr =
      new Attr(5); }
  Attr *attr;
};
```

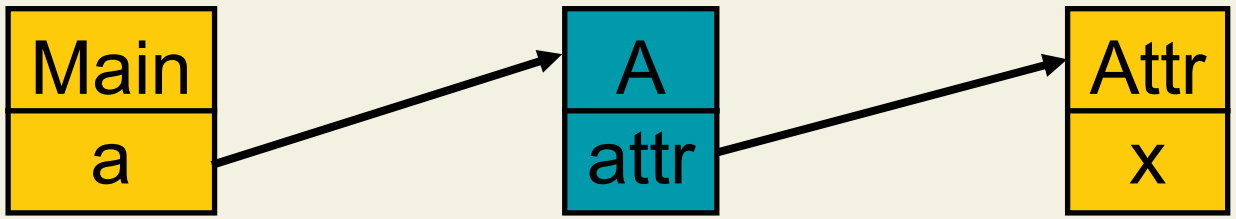
```
class Main {
  Main() { a =
        new A(); }

  const A *a;
};
```

C++ `const` is not transitive

```
Main *var = new Main();

var->a->attr->x = 9;
var->a->attr = new Attr(9); Compilation Error!
var->a = new A();
```



Readonly Access in Java

- Can be explicitly modeled by using read-only super-interfaces for all classes
- Only return read-only interface to clients
- Problems:
  - Not transitive, we need to change all involved interfaces
  - Hard to reuse existing libraries
  - Not safe, the representation can still leak and it is easy to cast the reference to the read-write type

Programming Discipline

- The programming discipline was just comments for the programmer
- No enforcement by the compiler
- No support for multiple developers
- Easy to make errors even for one developer

Extended Types as a Solution

- We extend the type system of our language to support additional modifiers on types
- Not just annotations in comments, but real changes to the language, new keywords and a different type system
- This provides us with direct support from the compiler in checking whether programs are correct
- We have an experimental Java compiler with support for extended types → not standard Java
- Today we discuss read-only types in detail

Types

- Each class or interface `T` **introduces two types**
- **Ground type** *ground(T)*
  - Denoted by `T` in programs
- **Readonly type** *ro(T)*
  - Denoted by **`readonly`** `T` in programs
- For each variable, attribute and parameter we can decide whether it should be read-write or read-only

Subtype Relation

- **Subtyping** among ground and readonly types is **defined as in Java**
  - S extends or implements T
    $\Rightarrow$ *ground(S)* < *ground(T)*
  - S extends or implements T
    $\Rightarrow$ *ro(S)* < ro*(T)*
- **Ground types** are **subtypes of** corresponding **readonly types**
  - *ground(T)* < *ro(T)*

Type Scheme Combinator

- Usually, an access of the form `a.b` has the type of `b`
- Now we need to combine the extended types of `a` and `b` to get the type of the result

a * b	*ground(T)*	*ro(T)*
ground(S)	*ground(T)*	*ro(T)*
ro(S)	*ro(T)*	*ro(T)*

- By using this combinator readonly becomes transitive

When do we need the combinator?

- When accessing attributes of an object:

```
class A:  B b;
a.b       ➔   result has type a * b
```

- When invoking methods:

```
class A:  B m();
a.m()     ➔   result has type a * B
```

Examples – Attribute Access

```
class A {       readonly B b;
                C c;
}
                A a1;
      readonly  A a2;
```

a * b	ground(T)	ro(T)
ground(S)	ground(T)	ro(T)
ro(S)	ro(T)	ro(T)

`a1.b` ➔ result type: `readonly B`

`a1.c` ➔ result type: `C`

`a2.b` ➔ result type: `readonly B`

`a2.c` ➔ result type: `readonly C`

Examples – Method Calls

```
class A { functional readonly B m1();
    functional B m2();
}
```

a * b	ground(T)	ro(T)
ground(S)	ground(T)	ro(T)
ro(S)	ro(T)	ro(T)

```
A a1;
readonly A a2;
```

`a1.m1()` ➔ result type: **`readonly`** `B`

`a1.m2()` ➔ result type: `B`

`a2.m1()` ➔ result type: **`readonly`** `B`

`a2.m2()` ➔ result type: **`readonly`** `B`

Functional Methods

- Tag side-effect free methods as **`functional`**
- Functional methods
  - Must not contain writing attribute access
  - Must not invoke non-functional methods
  - Must not create objects
  - Can only be overridden by functional methods
- Functional methods are necessary to allow method calls on a readonly reference
- Allowing all method calls on a readonly reference would allow anyone to change the readonly object through a method call

Examples – Functional Methods

```
class A {
  functional int sum() { return x+y; }
  functional readonly B getB() {
      return b; }
  functional int total() {
    int ltotal = 0;
    for( int i=0; i<arr.length; ++i )
      ltotal += arr[i];
    return ltotal;
  }
}
```

Examples – Attribute Access

```
class A {   readonly B b;
            C c;      }
A a1; readonly A a2;
B b1; readonly B b2;
C c1; readonly C c2;
```

a * b	ground(T)	ro(T)
ground(S)	ground(T)	ro(T)
ro(S)	ro(T)	ro(T)

```
b1 = a1.b;          B = ro B  → Invalid!
c2 = a1.c;          ro C = C  → Valid!
b2 = a2.b;       ro B = ro B  → Valid!
c1 = a2.c;          C = ro C  → Invalid!
```

Examples – Method Calls

```
class A {  functional readonly B m1();
  functional B m2(); }
A a1; readonly A a2;
B b1; readonly B b2;
```

a * b	ground(T)	ro(T)
ground(S)	ground(T)	ro(T)
ro(S)	ro(T)	ro(T)

```
b1 = a1.m1();        B = ro B  → Invalid!
b2 = a1.m2();        ro B = B  → Valid!
b1 = a2.m1();        B = ro B  → Invalid!
b2 = a2.m2();     ro B = ro B  → Valid!
```

Examples – Functional Methods

```
class A { int sum;
  functional int sum() {
    this.sum = x + y;
    return this.sum;
  }
}
class B extends A {
  int sum() {…}
}
```

Modification of Attribute!

Overriding functional method with non-functional!