UNIVERSITY OF
CAMBRIDGE
Computer Laboratory

Computer Science Tripos Part IB

Concepts in Programming Languages

http://www.cl.cam.ac.uk/Teaching/1112/ConceptsPL/

Alan Mycroft am@cl.cam.ac.uk

2011–2012 (Easter Term)

Additional notes

The notes contain material which I regard as important (and certainly part of the syllabus). The material was lectured, but did not appear on the course hand-out slide copies, so I reproduce it here to help with revision.

Alan Mycroft 22/5/2012

Access qualifiers and encapsulation

One of the key ideas of object-oriented languages is *abstraction* – that internal details of an object can be hidden from the rest of the system. Another word for this is that objects are *encapsulated*.

Is it tempting to believe that access qualifiers (private, protected and public) provide encapsulation. However, in one respect they fail to do this in that access qualifiers control access to *fields and methods*, whereas for reasoning about an object we wish to restrict access to the *object itself*. The difference is between controlling a reference to an object and the object itself (which may be accessed by an aliasing reference).

Two examples illustrate this (the former not lectured, but both friends):

```
class C
{   private int x;
    public C() { x = 0; }
    public void inc() { x++; }
    public void questionable(C other) { x--; other.x--; }
};
```

Question: should `questionable` be allowed to modify `other.x` as well as `this.x`? What does Java say?

The other example is:

```
class Bar { ... } ;
class Foo
{   private Bar mybar;
    public Foo() { mybar = new Bar; }
    ...
};
```

It is tempting to think that the `private Bar` encapsulates the use of its `Bar` object which may only be used for `Foo`-private purposes. (Such objects are often called '*rep*' (for representation) objects for `Foo` objects.) However, this would be untrue if `Foo` also contained

```
public Bar cheat() { return mybar; }
```

Now, presumably no-one would deliberately write such a clear security hole, but there are many other ways in which an *alias* to an object referenced by such a private field may be carried outside the object, for example returning an object which refers to a collection which refers to the value in `mybar`. Another example would be having an alternative constructor

```
public Foo(Bar x) { mybar = x; }
```

which is called as `Foo(y)` and where `y` is used again after the call to `Foo`, so the object in private `mybar` once again has a non-private alias held in the variable `y`.

Access qualifiers only check static uses; they do not do a proper job of enforcing encapsulation, e.g. that the object pointed to my `mybar` has no aliases.

To give a practical example, suppose my `Car` object has fields containing `Brakes`, `Engine` and `CPU` objects. Not only should these fields be private, but I would wish that `Car` objects should only be able to manipulate their *own* `Brakes`, `Engine` and `CPU` objects.

ML polymorphism

Type inference[1] can in general be applied to any language, and indeed C♯ has a form of type inference for initialised variables (because its types can become very verbose).

However, inference is more useful for some languages than others. For example in a language with Algol-like or Java-like types, but expressed in ML-style syntax, we can deduce:

```
⊢ fn x => x+1 : int -> int
⊢ fn x => not x : bool -> bool
⊢ fn x => x : int -> int
⊢ fn x => x : bool -> bool
```

For the first two examples, the inference is simple – the expression has a single type. For `fn x => x` the situation is more complicated as we can infer two distinct types – neither better than the other – and guessing the 'wrong' one will produce mysterious type errors elsewhere.

The key idea of ML type inference is to enrich the *language's* type system with type variables (α etc.) *within* the type system itself. So instead of saying (as in the example above)

⊢ `fn x => x :` t `->` t

for all non-polymorphic types t, we now say, *in addition to* the other judgements for `fn x => x`, that

⊢ `fn x => x :` α `->` α

or even

⊢ `fn x => x :` $\forall\alpha.\ \alpha$ `->` α

This type is now *more general* than all other types which can be inferred for `fn x => x`, and is known as the *principal type*. See the Part II Types course for more details.

One last point is that the ML type system works very well for pure functional languages (both lazy and eager). However, impure constructs can cause complications in the type system. The slides identify that *some* polymorphic exceptions, e.g.

```
exception Poly of 'a ;   (*** ILLEGAL!!! ***)
(raise Poly true) handle Poly x => x+1 ;
```

can give type-unsafe programs unless faulted, but assignable variables cause similar problems. Consider where the error(s) should be in the following program which, if executed, would be type-unsafe by constructing a list containing an integer and a boolean:

```
val x = ref [] : ('a list) ref;
x := 3 :: (!x);
x := true :: (!x);
print x;
```

There are multiple treatments of types which reject such programs in different places, but Standard ML chooses to fault the first line. Why? What is the error message?

[1] Many authors now prefer the phrase *type reconstruction* suggesting that the program was originally properly typed (so from a universe of Platonic programs if one has philosophical leanings) but some action has caused these types to be wholly or partly erased from the program.

Variance

While the notes cover all the formal material, I thought it would be useful to mention variance in Java. We all understand Java subtyping:

```
class Fruit { ... }
class Apple extends Fruit { ... }
class Banana extends Fruit { ... }
Fruit f; Apple a; Banana b;
// each of the following lines should be viewed in isolation
f = a; // OK
f = b; // OK
a = b; // compile-time error
a = f; // compile-time error
a = (Apple)f; // OK, but possible run-time error.
```

However, now consider generic types: Java built-in arrays, and Java generics (added later to the language) such as `ArrayList<...>` from the standard library. Now consider

```
Fruit[] f_ar; Apple a_ar[] = new Apple[10];
f_ar = a_ar;  // allowed in Java (perhaps a design mistake)
f_ar[0] = b;  // type checks OK, but run-time exception
// note this program must fail, because otherwise a_ar[0]
// would be a Banana!
```

We say Java arrays are *covariant* because `a_ar` is considered a subtype of `f_ar` (which parallels the way that `Apple` is a subtype of `Fruit`). By contrast the generic type `ArrayList` (from the Java library) is invariant (or non-variant), hence the following code is rejected by the compiler (because the two instances of `ArrayList` are incompatible).

```
ArrayList<Fruit> f_al; ArrayList<Apple> a_al;
f_al = a_al;  // compile-time error
```

Java *does* have the ability to assign an `ArrayList<Apple>` to an `ArrayList<Fruit>`, but only by forbidding array update operations on the latter (which were the source of the problems above). We instead write (using Java 'wildcards'):

```
ArrayList<? extends Fruit> f_al; ArrayList<Apple> a_al;
f_al = a_al;  // OK
a_al.get(3); OK -- returns type Apple
f_al.get(3); OK -- returns type Fruit
a_al.put(3, new Apple());  // OK
f_al.put(3, new Apple());  // write gives compile-time error
```

Remark: the underlying issues about co- and contra-variance can also be seen in the previous section on ML and polymorphic exceptions and polymorphic `ref` types.

Parallel Languages

Ideally, this course should have a lecture on parallel languages to include the core ideas of: Erlang, Cilk, X10, OpenMP and the like. I encourage you to read up on such languages (which lie outside the current syllabus).