# Cubex with Anonymous Classes

Ross Tate

November 26, 2013

## 1 Anonymous Classes

### 1.1 Syntax

We augment the Cubex grammar with the below production rule for expressions, which simultaneously declares and instantiates an anonymous class.

$$e ::= \cdots \mid \tau(e, \ldots, e)\{\textbf{fun}\ \nu_v\sigma\ s \ldots \textbf{fun}\ \nu_v\sigma\ s\}$$

### 1.2 Typechecking

We need new rules to typecheck these expressions. The formal rules are below, but the idea is simple:

- $\tau$ must be a valid type.
- All variables in scope in the context surrounding the anonymous class are immutable variables in the scope of the body of the anonymous class.
- The body of the anonymous class is typechecked as if $\tau$ were its declared super type.

Here are the formalities. We add the following rule for the judgement $\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e : \tau$.

$$\frac{\Psi \mid \Delta \mid \Gamma \vdash \textbf{class}\ \nu\langle\Theta\rangle()\ \textbf{extends}\ \tau\ \{\textbf{super}(e_1, \ldots, e_k);\ \textbf{fun}\ \nu_1\sigma_1\ \hat{s}_1\ \ldots\ \textbf{fun}\ \nu_n\sigma_n\ \hat{s}_n\} : \Psi' \mid \Delta'}{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash \tau(e_1, \ldots, e_k)\{\textbf{fun}\ \nu_1\sigma_1\ \hat{s}_1\ \ldots\ \textbf{fun}\ \nu_n\sigma_n\ \hat{s}_n\} : \tau}$$

In other words, if one could make a generic class $\nu\langle\Theta\rangle$ inheriting $\tau$ with that body that would be accepted by the type checker, then the expression is valid. Note that the name $\nu$ is not used elsewhere, hence the term "anonymous".

### 1.3 Semantics and Implementation Hints

At runtime, an expression which declares and instantiates an anonymous class evaluates to a new object whose methods are as declared in the body of the anonymous class. The new object captures the state of the variables in its surrounding context at instantiation time. That is, if a mutable variable in the surrounding context is updated after the anonymous class is instantiated, this is invisible to the new object. For example:

```
interface Foo {
  fun getStr() : String;
}
s := "hello";
a := Foo() {
  fun getStr() : String {
    return s;
  }
}
s := "world";
return [a.getStr()];
```

will return an iterable with the single element `"hello"`.

One possible way to implement anonymous classes is as follows. In your parser, add a new AST node for the new expression production rule. Modify your typechecker to handle typechecking for anonymous classes. Either in your typechecker or afterwards, analyze which variables of the surrounding scope are used in each anonymous class. Then add a pre-optimization pass which lifts each anonymous classe to a named top-level classe whose constructor takes as parameters the variables from the surrounding contexted which the anonymous class uses. Replace the declaration of the anonymous class with a call to the constructor of the new named class. Using this implementation strategy, you should not need to change your optimization or code generation phases.

## 1.4 Grading

In grading this extension, we will place the vast majority of grading weight on tests where the anonymous classes have single, non-generic inheritance from a class or interface. We will place small amount of weight on generics and multiple inheritance.