Elements of Programming Languages
Assignment 3: You, too, can write an object-oriented programming language!
Version 1.2 (Updated November 15)
Due November 21 at 4pm

1 Introduction

In Lectures 8–11 we have covered the general outlines of modules, interfaces, objects, and classes. In this assignment, you will put several of these elements together and implement the key parts of a simple, but realistic object-oriented language, which we will call Elephant.

Elephant provides most of the key features of a pure, typed, object-oriented language, similar to Scala but without traits or polymorphism (or many other features). These include:

- First class objects: Objects can be created (as values) directly using the `object` construct, or indirectly from a class using the `new` keyword. Objects can contain fields and methods (i.e. fields whose values are functions); moreover, fields or methods can be updated, resulting in a new object.
- Classes: Classes can also be created (as values) using the `class` construct. Classes can be defined as subclasses of other classes, and subclasses can add new methods or override existing methods.
- Types and subtyping: Objects and classes are typed, and the type associated with a subclass must be a subtype of the superclass type.

In addition, Elephant has many basic constructs familiar from Giraffe, and provides multi-argument functions and application and type abbreviations.

We will implement Elephant by translating its constructs to a simpler, core language, called Core Elephant. Core Elephant is an untyped calculus similar to $\mathsf{L}_{\mathsf{Lam}}$, extended with a primitive *object* construct and associated field dereferencing and field update operations. Like the Elephant source language, Core Elephant is *pure*, that is, there are no references or side effects. Despite its simplicity, Core Elephant is Turing-complete and expressive enough to represent all of the features of Elephant using a desugaring translation. Thus, the first two parts of the assignment will focus on implementing substitution and evaluation for Core Elephant and the second half on typechecking source Elephant programs and then desugaring them to Core Elephant for execution.

Each part of this assignment is described in greater detail in the rest of this handout. Although there are some dependencies between these exercises, you may find it helpful to do the easier parts of each exercise before proceeding to the more demanding parts.

This assignment is due November 21 at 4pm.

Please read over this handout carefully and look over the code before beginning work, as some of your questions may be answered later. Please let us know if there are any apparent errors or bugs. We will try to update this handout to fix any major problems and such updates will be announced to the course mailing list. The handout is versioned and the most recent version should always be available from the course web page.

2 Getting started

`Assn3.zip` contains a number of starting files; use the unzip command to extract them. We provide the following Scala files.

- `Core.scala`: contains the core language abstract syntax and skeleton code for the substitution and evaluation functions. You will modify this to do the exercises in Section 3.
- `Source.scala`: contains the abstract syntax for Elephant and skeleton code for the typechecker. You will modify this to do the exercises in Section 4.
- `SourceHelpers.scala`: contains code to help with typechecking Elephant, including a subtyping function. You should not (need to) modify this.
- `Desugar.scala`: contains a skeleton for the desugaring function. You will modify this to do the exercises in Section 5.
- `Assn3.scala`: contains the parser and main function. You should not (need to) modify this.
- `Utility.scala`: contains some helper functions, including `Gensym.gensym` that was used in Assignment 2 to generate fresh variables.

We also include several example programs that illustrate Elephant and Core Elephant syntax.

We also provide several scripts (which should work on a DICE, Linux or MacOS system):

- `compile.sh` compiles the code
- `clean.sh` cleans up the directory, removing any compiled code
- `run.sh` runs the interpreter. The command line arguments `-core` or `-source` choose whether to execute a Core Elephant or source Elephant program. The last argument is interpreted as the name of a file to run.
- We also provide (in the `tests` directory) a test suite which you can use to test your solutions. This covers the most common cases of the core evaluator, the typechecker, and the desugaring pass.

  These tests make use of `scalatest`. We have included a version of `scalatest.jar` which will work on the DICE environment (Scala 2.11).

  To run the test suite, run `./compile.sh`, which will compile your solution and the test suite. You can then run the test suite using `./runTests.sh`.

  Note that the tests cover the most common cases, but we do not guarantee that a solution which passes all of the tests will get a perfect score. A solution which passes all the tests probably wouldn't do badly, however!

If you are using a non-DICE system, such as Windows, these scripts may not work, but you can look at the contents to work out what you need to do instead.

Finally, we provide a JAR files that contains a sample solution `Assn3SampleSolution.jar`. You can run this as follows:

```
$ scala -cp ".:spc.jar" Assn3SampleSolution.jar -core <filename>  # runs the interpreter on a core file
$ scala -cp ".:spc.jar" Assn3SampleSolution.jar -source <filename>  # runs the interpreter on a source file
```

2.1 Objectives

The rest of this handout defines exercises for you to complete, building on the partial implementation in the provided files. You may add your own function definitions or other code, but please use the existing definitions/types for the functions we ask you to write in the exercises, to simplify automated testing we may do. You should not need to change any existing code other than filling in definitions of functions as stated in the exercises below.

Your solutions may make use of Scala library operations, such as the list and list map operations that have been covered in previous assignments.

This assignment relies on material covered up to Lecture 11 (October 30).

This assignment is graded on a scale of 100 points, and amounts to 25% of your final grade for this course. Your submissions will be marked and returned with feedback within 2 weeks if they are received by the due date. Late submissions (submitted within 7 days of the deadline) will incur a penalty in line with the School policy on late submissions, and will be returned with feedback within 2 weeks of the time of submission. **Please let us know before the deadline if you intend to submit late.**

Unlike the other two assignments, which were for feedback only, you must work on this assignment individually and not with others, in accordance with University policy on academic conduct. Please see the course web page for more information on this policy.

Submission instructions You should submit a single ZIP file, called `Assn3.zip`, with the missing code in the three main Scala files filled in as specified in the exercises in the rest of this handout. To submit, use the following DICE commands:

```
$ zip Assn3.zip Core.scala Source.scala Desugar.scala
$ submit epl cw3 Assn3.zip
```

The submission deadline is 4pm on November 21.

3 Core Language

3.1 Syntax

Figure 1 shows the abstract syntax of Core Elephant. Most of the constructs should be familiar, as they are a subset of the Giraffe language covered in Assignment 2.

The new constructs are *objects*, *field dereference* and *field update*. The object construct is of the form:

$$[\{\ell_i = (x_i)\ \texttt{=>}\ e_i\}_i]$$

Here, the notation $\{\ell_i = (x_i)\ \texttt{=>}\ e_i\}_i$ indicates that an object may contain a (semicolon-separated) sequence of field definitions, each of the form $\ell = (x)\ \texttt{=>}\ e$. So, concretely, an object expression could be of the form

```
[a = (x) => 1; b = (x) => \y. x.a + y]
```

This defines an object with two fields, `a` and `b`. The first field `a` is simply a number, while the second is a method that adds the first field's value to its argument. In each case `x` is the *self parameter*, which we do not use in the definition of `a`, but in the definition of `b` we do need to use it to access the object's `a` field.

Fields of objects can be accessed similarly to those of records using the syntax $e.\ell$; if we take o to be the object defined above then $o.a$ should evaluate to 1 and $o.b(42)$ should evaluate to 43. Furthermore, we allow field *update* expressions of the form $\texttt{update}\, e_1 :: \ell\, \texttt{with}\, ((x)\ \texttt{=>}\ e_2)$. This says to update the object value of e_1 so that the ℓ field is redefined as $(x)\ \texttt{=>}\ e_2$. Since Core Elephant is a pure language, this does not change the existing value of e_1 but returns a *new object* that is identical to e_1 except that the ℓ field has been redefined. For example,

$$(\texttt{update}\, o :: a\, \texttt{with}\, (x)\ \texttt{=>}\ 17).b(42)$$

should evaluate to $17 + 42 = 59$.

Expressions	e	::=	x	Variables
		|	$\lambda x.e$	Lambda abstraction
		|	$e_1(e_2)$	Function application
		|	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	Let-binding
		|	$n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1/e_2$	Numbers and Arithmetic operations
		|	$\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$	Conditional
		|	$\mathsf{true} \mid \mathsf{false} \mid !e \mid e_1\ \&\&\ e_2 \mid e_1\ \|\|\ e_2 \mid e_1 == e_2$	Booleans and equality testing
		|	$s \in \mathit{String}$	Strings
		|	$[\{\ell_i = (x_i)\ \texttt{=>}\ e_i\}_i]$	Objects
		|	$e.\ell$	Field / method selection
		|	$\mathsf{update}\ e_1 :: \ell\ \mathsf{with}\ ((x)\ \texttt{=>}\ e_2)$	Field / method update
Values	v	::=	$\lambda x.e$	Lambda abstraction
		|	$n \in \mathbb{N}$	Numbers
		|	$\mathsf{true} \mid \mathsf{false}$	Booleans
		|	$s \in \mathit{String}$	Strings
		|	$[\{\ell_i = (x_i)\ \texttt{=>}\ e_i\}_i]$	Objects

Figure 1: Syntax of the core language

$$
\begin{array}{rcl}
v[e/x] & = & v \\
x[e/x] & = & e \\
y[e/x] & = & y \qquad \text{if } x \neq y \\
(e_1 \oplus e_2)[e/x] & = & e_1[e/x] \oplus e_2[e/x] \\
(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)[e/x] & = & \mathsf{if}\ e_1[e/x]\ \mathsf{then}\ e_2[e/x]\ \mathsf{else}\ e_3[e/x] \\
(!e_1)[e/x] & = & !(e_1[e/x]) \\
(\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2)[e/x] & = & \mathsf{let}\ z = e_1[e/x]\ \mathsf{in}\ (e_2(y \leftrightarrow z))[e/x] \qquad z \text{ is fresh} \\
(\lambda y.e_1)[e/x] & = & (\lambda z.(e_1(y \leftrightarrow z))[e/x]) \qquad z \text{ is fresh} \\
(e_1(e_2))[e/x] & = & e_1[e/x](e_2[e/x]) \\
[\{\ell_i = (y_i)\ \texttt{=>}\ e_i\}_{i \in 1..n}][e/x] & = & [\{\ell_i = (z_i)\ \texttt{=>}\ (e_i(y_i \leftrightarrow z_i))[e/x]\}_{i \in 1..n}] \qquad z_i \text{ are fresh} \\
(e_1.\ell)[e/x] & = & (e_1[e/x]).\ell \\
(\mathsf{update}\ e_1 :: \ell\ \mathsf{with}\ ((y)\ \texttt{=>}\ e_2))[e/x] & = & \mathsf{update}\ (e_1[e/x]) :: \ell\ \mathsf{with}\ ((z)\ \texttt{=>}\ (e_2(y \leftrightarrow z))[e/x]) \qquad z \text{ is fresh}
\end{array}
$$

Figure 2: Capture-avoiding Substitution for Core Language

$$
\frac{}{v \Downarrow v} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2[v_1/x] \Downarrow v_2}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v_2} \qquad \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_1 \qquad e[v_1/x] \Downarrow v}{e_1(e_2) \Downarrow v}
$$

$$
\frac{e_1 \Downarrow v \qquad e_2 \Downarrow v}{v \text{ is an integer, boolean, or string}}{e_1 == e_2 \Downarrow \mathsf{true}} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \neq v_2}{v_1, v_2 \text{ are integers, booleans, or strings}}{e_1 == e_2 \Downarrow \mathsf{false}}
$$

$$
\frac{e \Downarrow \mathsf{true} \qquad e_1 \Downarrow v_1}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v_1} \qquad \frac{e \Downarrow \mathsf{false} \qquad e_2 \Downarrow v_2}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v_2} \qquad \frac{\oplus \in \{+, -, *, /\} \qquad e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \oplus e_2 \Downarrow v_1 \oplus_{\mathbb{N}} v_2}
$$

$$
\frac{e \Downarrow v}{!e \Downarrow \neg v} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1\ \&\&\ e_2 \Downarrow v_1 \wedge v_2} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1\ ||\ e_2 \Downarrow v_1 \vee v_2} \qquad \frac{\begin{array}{c} v_1 = [\{\ell_i = (x_i)\ \texttt{=>}\ e_i\}_{i \in 1..n}] \\ e \Downarrow v_1 \qquad j \in 1..n \qquad e_j[v_1/x_j] \Downarrow v_2 \end{array}}{e.\ell_j \Downarrow v_2}
$$

$$
\frac{\begin{array}{c} v_1 = [\{\ell_i = (x_i)\ \texttt{=>}\ e_i\}_{i \in 1..n}] \\ e \Downarrow v_1 \qquad v_2 = v_1\ \mathsf{with}\ \ell_j = ((x)\ \texttt{=>}\ e') \end{array}}{\mathsf{update}\ e :: \ell_j\ \mathsf{with}\ ((x)\ \texttt{=>}\ e') \Downarrow v_2}
$$

Figure 3: Big-step semantics of the core language

3.2 Substitution

Figure 2 shows the definition of substitution for Core Elephant. The cases for familiar constructs are straightforward. The cases for objects, field dereferences and field updates are new; in each case, note that in subexpressions of the form (x) => e, the variable x is bound in e, so needs to be renamed.

Exercise 1. *In `Core.scala`, implement the Core Elephant substitution function:*

```
def subst(e1: Expr, e2: Expr, x: Variable): Expr
```

[20 marks]

3.3 Operational Semantics

Figure 3 presents the large-step semantics of Core Elephant. First, observe that object expressions are values (hence, no further evaluation is needed).

In the rule for evaluating object field references $e.\ell$, we first evaluate e to a value v, which has to be an object containing a field $\ell = (x)$ => e_ℓ. We then evaluate e_ℓ after substituting v for x. This is because the self parameter x of ℓ needs to be replaced with the actual object value; one should compare this with the rule for applying a recursive function from Lecture 5.

The rule for evaluating an expression update e_1 :: ℓ with (x) => e_2 updating an object value proceeds along similar lines. First, we evaluate e_1 which must yield an object value v_1. We then construct a new object value v_2 obtained by replacing the ℓ field of v_1 with (x) => e_2, and this object value is the result.

Exercise 2. *In `Core.scala`, implement the Core Elephant evaluation function `eval`:*

```
def eval(e: Expr): Value
```

[30 marks]

4 Elephant Source Language

Figure 4 shows the syntax of the Elephant source language.

Expression constructs. The Elephant source language contains let-bindings, numbers and arithmetic, conditional expressions, Booleans and logical operations, strings, and equality. These are identical to the core language.

Multi-argument functions. In the source language, we opt (like Scala) to include multi-parameter functions, and multi-argument function application. This allows us to write expressions such as the following:

$$\begin{array}{l}\mathsf{let}\ \mathit{addTwo} = \mathsf{fun}(x : \mathsf{Int}, y : \mathsf{Int})\{x + y\}\ \mathsf{in}\\ \mathit{addTwo}(10, 15)\end{array}$$

which will in turn evaluate to 25.

Type synonyms. Since Elephant requires type annotations on functions, objects, and classes, we also implement a notation for type synonyms. For example, it is possible to write

$$\begin{array}{l}\mathsf{type}\ \mathit{Colour} = \mathsf{Object}(X)[\mathtt{red} : \mathsf{Int}; \mathtt{green} : \mathsf{Int}; \mathtt{blue} : \mathsf{Int}]\ \mathsf{in}\\ \mathsf{fun}(c : \mathit{Colour})\,\{\,c\,\}\end{array}$$

Writing programs without type synonyms would be rather painful in this setting, since it would be necessary to repeat the object type everywhere! Nonetheless, the mechanics of resolving the

Types	τ	::=	X	Type Variables
		|	$\mathsf{Object}(X)[\{\ell_i : \tau_i\}_{i \in 1..n}]$	Object type
		|	$\mathsf{Class}(\tau)$	Class type
		|	$(\tau_1, \ldots, \tau_n)$ -> τ	Function type
		|	String | Bool | Int	Base type
Expressions	e	::=	x	Variables
		|	$\mathsf{object}(x : \tau)\,\{\{\ell_i = e_i\}_{i \in 1..n}\}$	Objects
		|	root	Root class
		|	$\mathsf{class}(x : \tau)\,\{\{\ell_i = e_i\}_{i \in 1..n}\}$	Classes
		|	$\mathsf{class}(x : \tau_1)\ \mathsf{extends}(e : \tau_2)\,\{$ $\{\ell_i^{\mathsf{dec}} = e_i\}_{i \in 1..m}$ $\}\ \mathsf{overrides}\,\{$ $\{\ell_j^{\mathsf{ovr}} = e_j\}_{j \in 1..n}$ $\}$	Classes with inheritance
		|	new e	Object construction from a class
		|	$\mathsf{fun}(x_1 : \tau_1, \ldots, x_n : \tau_n)\,\{e\}$	Functions
		|	$e(e_1, \ldots, e_n)$	Function application
		|	$e.\ell$	Field selection / method invocation
		|	$e_1.\ell := e_2$	Field update
		|	$e_1.\ell := \mathsf{method}(x : \tau)\{e_2\}$	Method update
		|	type $X = \tau$ in e	Type synonyms
		|	let $x = e_1$ in e_2	Let-binding
		|	$n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2$	Numbers and Arithmetic operations
		|	if e then e_1 else e_2	Conditional
		|	true | false |$!e \mid e_1 \,\&\&\, e_2 \mid e_1 \,\|\|\, e_2 \mid e_1 == e_2$	Booleans and equality testing
		|	$s \in \mathit{String}$	Strings

Figure 4: Syntax of the Elephant Source Language

aliases are uninteresting for the purposes of the assignment, and we have handled this for you. **Type synonyms are resolved for you in a preprocessing step. You should never see a `TypeIn` construct in a program, so you can safely ignore it in any passes that you write.**

Objects. Objects in the Elephant source language are similar to objects in the source language, except that there is a single *self* parameter for all fields in the object.

For example, the object from Section 3 could be written:

$$
\begin{array}{l}
\mathsf{type}\ \mathit{ObjTy} = \mathit{Object}(X)[a : \mathsf{Int}; b : (\mathsf{Int}) \to \mathsf{Int}]\ \mathsf{in} \\
\mathsf{object}(\mathit{this} : \mathit{ObjTy})\ \{ \\
\quad a = 1; \\
\quad b = \mathsf{fun}(y : \mathsf{Int})\{(\mathit{this}.x) + y\} \\
\}
\end{array}
$$

Note that since the source Elephant language (unlike the core language) is typed, that we need an explicit type annotation on the object's *self* parameter. The typing rules ensure that the object definition conforms to its type.

Projection of an object's field is exactly the same as in Core Elephant. Again calling the object o, we can access field a using the notation $o.a$.

On the other hand, we need a slightly different notation for updating fields and methods as we are assuming that the same *self* parameter is used for all fields.

The notation

$$e_1.\ell := e_2$$

assigns the value of e_2 to the field ℓ of e_1, returning the updated object. Note that since a field does not use its *self* parameter, that there is no need to specify it explicitly.

The notation

$$e_1.\ell := \mathsf{method}(x : \tau)\{\, e_2 \,\}$$

denotes method update and assigns the value of e_2 to the field ℓ of object e_1, retuning the updated object. Unlike field update, e_2 can use a *self*-parameter x of type τ. The typing rule for method update ensures that τ is the same type as the type of the object.

An object type is written

$$\mathsf{Object}(X)[\{\ell_i : \tau_i\}_{i \in 1..n}]$$

meaning an object with fields ℓ_i of types τ_i. Note also that the type contains a *type variable* X, which makes it possible to write methods which refer to values of the type being defined. As an example, consider the following object:

$$\begin{array}{l}
\mathsf{type}\ \mathit{EqNum} = \mathsf{Object}(X)[\mathtt{num} : \mathsf{Int}, \mathtt{eq} : (X)\ \texttt{->}\ \mathsf{Bool}]\ \mathsf{in} \\
\mathsf{object}(\mathit{this} : \mathit{EqNum})\{ \\
\quad \mathtt{num} = 10; \\
\quad \mathtt{eq} = \mathsf{fun}(\mathit{other} : \mathit{EqNum})\{\, \mathit{this}.\mathtt{num} == \mathit{other}.\mathtt{num} \,\} \\
\}
\end{array}$$

Subtyping. Elephant uses *structural* subtyping: that is, an object o_1 is a subtype of an object o_2 if o_1 has the same fields as o_2, plus (optionally) some additional ones. For reasons of soundness, fields that are common between o_1 and o_2 must have the same types. More specifically, we support *width* subtyping, but not *depth* subtyping.

Luckily for you, we have implemented subtyping. You therefore don't need to worry about implementing the subtyping algorithm yourself, but only using the subtyping function `subtypeOf` where necessary as stated by the rules.

Classes. A *class* can be thought of as a "template" for creating an object. Classes provide the possibility for subtyping and inheritance, which are not supported by plain objects.

A "plain" class which does not inherit from a superclass can be written:

$$\mathsf{class}(x : \tau)\,\{\{\ell_i = e_i\}_{i \in 1..n}\}$$

where x is the name of the self-binder in the class, and τ is the type of object constructed by the class. Again, $\{\ell_i = e_i\}_{i \in 1..n}$ denotes a list of fields with names ℓ_i and bodies e_i, which can all use name x.

Classes have type $\mathsf{Class}(\tau)$, which can be read as "a class able to construct objects of type τ." Given a class c, we can write $\mathsf{new}\, c$ to use the class to create an object.

We can also use classes to support subtyping and inheritance. Consider the following program:

```
type Point = Object(X)[x : Int; y : Int] in
type AnnotatedPoint = Object(X)[x : Int; y : Int; label : String] in
let pointClass = class(this : Point) {x = 100; y = 0} in
let annotatedPointClass =
    class(this : AnnotatedPoint) extends(pointClass : Class(Point)) {
      label = "origin"
    } overrides {
      x = 0
    } in
new annotatedPointClass
```

We begin by defining two object types, *Point* and *AnnotatedPoint*. A point consists of an x and a y field, and an *AnnotatedPoint* also contains a label, which is a string.

We next define a class *pointClass*, which constructs objects of type *Point*, with the x co-ordinate initialised to 100 and the y co-ordinate initialised to 0.

Next, we define a class *annotatedPointClass* which extends the *pointClass* type to create *AnnotatedPoint* objects. In particular, *annotatedPointClass* defines the label field to be initialised to "origin", and *overrides* the definition of the x co-ordinate to be 0. Since y is unspecified, it *inherits* the initial value 0 from *pointClass*.

Finally, we create an object from *annotatedPointClass* using new.

4.1 Typing rules

We describe the rules in three sections: the functional features; object-oriented features; and class-based features.

Helper functions. In this section, we will guide you through the implementation of a typechecker for Elephant. To do so, we have provided you with several helper functions:

- `def subtypeOf(ty1: Type, ty2: Type): Boolean` returns `true` if type `ty1` is a subtype of `ty2`, and `false` otherwise. In the formal rules, this function corresponds to $\tau_1 <: \tau_2$.
- `def equivTypes(ty1: Type, ty2: Type): Boolean` returns `true` if types `ty1` and `ty2` are α-equivalent (that is, equal up to naming of bound type variables) and `false` otherwise. In the formal rules, this function corresponds to $\tau_1 = \tau_2$.
- `def typeSubst(ty1: Type, ty2: Type, tyVar: TypeVariable): Type` substitutes `ty2` for free occurrences of `tyVar` in `ty1`. In the formal rules, this function corresponds to $\tau_1[\tau_2/X]$.

Functional rules. The rules for variables, let-bindings, values, operations, and conditional expressions are identical to those in Assignment 2. Note that we use schematic syntax to abstract over the integer operations.

Since Elephant supports multi-argument functions and multi-argument application, we require new rules. The rule for function definition states that given arguments x_i of types τ_i for $i \in 1..n$, if we extend the environment Γ with $\{x_i : \tau_i\}_{i \in 1..n}$ and deduce that the function body e has type τ, then the whole function has type $(\tau_1, \ldots, \tau_n)$ -> τ. This is similar to applying the rule for lambda-abstraction multiple times.

Function application is similar. Say we wish to typecheck a function application $e(e_1, \ldots, e_n)$. We must firstly check the type of the function e, which must be of function type $(\tau_1, \ldots, \tau_n)$ -> τ. Given this, we must check the type of each argument e_i, giving us types τ_i'. Finally, we must check that each argument τ_i' type is *a subtype of* the given parameter type τ_i. If this is the case, then the function application has the type τ.

Functional features

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \mathsf{fun}(x_1 : \tau_1, \ldots, x_n : \tau_n)\,\{e\} : (\tau_1, \ldots, \tau_n)\ \texttt{->}\ \tau}$$

$$\frac{\Gamma \vdash e : (\tau_1, \ldots, \tau_n)\ \texttt{->}\ \tau \quad (\Gamma \vdash e_i : \tau_i')_{i \in 1..n} \quad (\tau_i' <: \tau_i)_{i \in 1..n}}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{s \text{ is a string literal}}{\Gamma \vdash s : \mathsf{String}} \qquad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash b : \mathsf{Bool}} \qquad \frac{\oplus \in \{+, -, *, /\} \quad \Gamma \vdash e_1 : \mathsf{Int} \quad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : \mathsf{Bool}}{\Gamma \vdash e_1\ \&\&\ e_2 : \mathsf{Bool}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : \mathsf{Bool}}{\Gamma \vdash e_1\ ||\ e_2 : \mathsf{Bool}} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\mathsf{Int}, \mathsf{Bool}, \mathsf{String}\}}{\Gamma \vdash e_1 == e_2 : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool}}{\Gamma \vdash !e : \mathsf{Bool}} \qquad \frac{\Gamma \vdash e : \mathsf{Bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}$$

Object-oriented Features

$$\frac{\tau = \mathsf{Object}(X)[\{\ell_i : \tau_i\}_{i \in 1..n}] \quad (\Gamma, x : \tau \vdash e_i : \tau_i[\tau/X])_{i \in 1..n}}{\Gamma \vdash \mathsf{object}(x : \tau)\,\{\{\ell_i = e_i\}_{i \in 1..n}\} : \tau}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau = \mathsf{Object}(X)[\{\ell_i : \tau_i\}_{i \in 1..n}] \quad j \in 1..n}{\Gamma \vdash e.\ell_j : \tau_j[\tau/X]}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \tau = \mathsf{Object}(X)[\{\ell_i : \tau_i\}_{i \in 1..n}] \quad j \in 1..n \quad \Gamma \vdash e_2 : \tau' \quad \tau' = \tau_j[\tau/X]}{\Gamma \vdash e_1.\ell_j := e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \tau = \mathsf{Object}(X)[\{\ell_i : \tau_i\}_{i \in 1..n}] \quad j \in 1..n \quad \Gamma, x : \tau \vdash e_2 : \tau' \quad \tau' = \tau_j[\tau/X]}{\Gamma \vdash e_1.\ell_j := \mathsf{method}(x : \tau)\{e_2\} : \tau}$$

Classes

$$\frac{\Gamma \vdash e : \mathsf{Class}(\tau) \quad \tau = \mathsf{Object}(X)[\{\ell_i : \tau_i\}_{i \in 1..n}]}{\Gamma \vdash \mathsf{new}\ e : \tau} \qquad \frac{}{\Gamma \vdash \mathsf{root} : \mathsf{Class}(\mathsf{Object}(X)[])}$$

$$\frac{\begin{array}{c}\tau <: \tau' \quad \Gamma \vdash e : \mathsf{Class}(\tau') \quad (\tau_k'^{\mathsf{inh}}[\tau'/Y] = \tau_k^{\mathsf{inh}}[\tau/X])_{k \in 1..p} \\ (\Gamma, x : \tau \vdash e_i^{\mathsf{dec}} : \tau_i^{\mathsf{dec}}[\tau/X])_{i \in 1..m} \quad (\Gamma, x : \tau \vdash e_j^{\mathsf{ovr}} : \tau_j^{\mathsf{ovr}}[\tau/X])_{j \in 1..n}\end{array}}{\Gamma \vdash \mathsf{class}(x : \tau)\ \mathsf{extends}(e : \mathsf{Class}(\tau'))\ \{\{\ell_i^{\mathsf{dec}} = e_i^{\mathsf{dec}}\}_{i \in 1..n}\}\ \mathsf{overrides}\ \{\{\ell_j^{\mathsf{ovr}} = e_j^{\mathsf{ovr}}\}_{j \in 1..m}\} : \mathsf{Class}(\tau)}$$

where

$$\tau = \mathsf{Object}(X)[\{\ell_i^{\mathsf{dec}} : \tau_i^{\mathsf{dec}}\}_{i \in 1..m} \cup \{\ell_j^{\mathsf{ovr}} : \tau_j^{\mathsf{ovr}}\}_{j \in 1..n} \cup \{\ell_k^{\mathsf{inh}} : \tau_k^{\mathsf{inh}}\}_{k \in 1..p}]$$
$$\tau' = \mathsf{Object}(Y)[\{\ell_j^{\mathsf{ovr}} : \tau'^{\mathsf{ovr}}_j\}_{j \in 1..n} \cup \{\ell_k'^{\mathsf{inh}} : \tau'^{\mathsf{inh}}_k\}_{k \in 1..p}]$$

Figure 5: Typing rules for the source language

We have implemented the subtyping algorithm for you: to check whether a type `ty1` is a subtype of a type `ty2`, write:

```
subtypeOf(ty1, ty2)
```

Note that we do not have a separate subsumption rule, instead opting to check subtyping in the rule for function application.

Exercise 3. *In `Source.scala`, implement the functional rules by filling in the cases for `Var`, `Func`, `Apply`, `LetIn`, `IfThenElse`, `Bool`, `Num`, `Str`, `BinOp`, and `NotOp` cases in the `typeCheck` function.*

[10 marks]

Object-oriented Rules. We now come to the rules for objects.

The first rule checks the well-typedness of the construction of an object $\mathsf{object}(x : \tau)\,\{\{\ell_i = e_i\}_{i\in 1..n}\}$.

The first premise of the rule states that τ should be an object type of the form $\mathsf{Object}(X)[\{\ell_i : \tau_i\}_{i\in 1..n}]$. Note that this states that the type and the object being constructed must have exactly the same set of labels. The second premise of the rule states that, given the self-variable x of type τ added to the environment, that each e_i must have the type τ_i as specified in the object type.

The second rule checks the well-typedness of field accesses $e.\ell_j$. We firstly need to check that e has an object type $\mathsf{Object}(X)[\ell_i : \tau_i]_{i\in 1..n}$. Secondly, we need to check that ℓ_j is present in the object's fields. The final type is τ_j, but with the object type τ substituted for the self-variable X. To perform this type-level substitution, use the `typeSubst` function which we have provided for you; to substitute a type `t2` for type variable X in type `t1`, write:

```
typeSubst(t1, t2, "X")
```

The third rule checks the well-typedness of field updates $e_1.\ell_i := e_2$. To check that this is well-typed, we firstly need to check that e_1 has some object type $\mathsf{Object}(X)[\ell_i : \tau_i]_{i\in 1..n}$ and that the object type contains the label ℓ_j with type τ_j. Next, we need to typecheck e_2, determining that it has some type τ'. The final step is to check whether τ' is equivalent to τ_j, with τ substituted for the type variable X.

Remember to check equivalence of types using `equivTypes` instead of structural equality, since we're considering types equal up to α-equivalence!

The final object-oriented rule is for method update: this is very similar to field update, except we add the self variable $x : \tau$ to the typing environment when checking the method body.

Exercise 4. *In `Source.scala`, implement the rules for objects by filling in the `Object`, `SelectField`, `FieldUpdate`, and `MethodUpdate` cases in the `typeCheck` function.*

[10 marks]

Class Rules. Finally, we can implement the rules for checking that classes are well-typed.

To typecheck $\mathsf{new}\, e$, we check that e has type $\mathsf{Class}(\tau)$, where τ is an object type: if so, then $\mathsf{new}\, e$ has type τ.

The root class always has type $\mathsf{Class}(\mathsf{Object}(X)[])$.

The rule for typing classes is more delicate. Let us begin by considering the simplest case, that of a "plain" class which does not inherit from any other class.

$$\frac{\tau = \mathsf{Object}(X)[\{\ell_i : \tau_i\}_{i\in 1..n}] \qquad (\Gamma, x : \tau \vdash e_i : \tau_i[\tau/X])_{i\in 1..n}}{\Gamma \vdash \mathsf{class}(x : \tau)\,\{\{\ell_i = e_i\}_{i\in 1..n}\} : \mathsf{Class}(\tau)}$$

You may notice that this is very similar indeed to the rule for creating an object! Here, we firstly check that the self-type τ is an object type, before checking that each e_i has the type τ_i corresponding to the object type. Finally, the whole expression has type $\mathsf{Class}(\tau)$, allowing the class to be used to construct new instances of an object with type τ.

Next, let us consider the case where we can *inherit* some of the methods from a superclass (but not override them, yet).

$$\frac{\tau <: \tau' \qquad \Gamma \vdash e : \mathsf{Class}(\tau') \qquad (\Gamma, x : \tau \vdash e_i^{\mathsf{dec}} : \tau_i^{\mathsf{dec}}[\tau/X])_{i \in 1..m} \qquad (\tau_k'^{\mathsf{inh}}[\tau'/Y] = \tau_k^{\mathsf{inh}}[\tau/X])_{k \in 1..p}}{\Gamma \vdash \mathsf{class}(x : \tau)\ \mathsf{extends}(e : \mathsf{Class}(\tau'))\ \{\{\ell_i^{\mathsf{dec}} = e_i^{\mathsf{dec}}\}_{i \in 1..m}\}\ \mathsf{overrides}\ \{\} : \mathsf{Class}(\tau)}$$

$$\text{where} \quad \tau = \mathsf{Object}(X)[\{\ell_i^{\mathsf{dec}} : \tau_i^{\mathsf{dec}}\}_{i \in 1..m} \cup \{\ell_k^{\mathsf{inh}} : \tau_k^{\mathsf{inh}}\}_{k \in 1..p}]$$
$$\tau' = \mathsf{Object}(Y)[\{\ell_k^{\mathsf{inh}} : \tau_k'^{\mathsf{inh}}\}_{k \in 1..p}]$$

Here, we distinguish the set of *declared* labels ℓ_i^{dec} from the set of *inherited* labels ℓ_k^{inh}. Since we are not allowing overriding yet, we know that the set of declared labels must be labels occurring in τ but not in τ'. To typecheck this, we firstly need to check whether $\tau <: \tau'$ (remember, use the provided `subtypeOf` function for this). We also need to check whether the superclass e has type $\mathsf{Class}(\tau')$.

To ensure soundness, we need ensure that the types of the inherited labels are equal to the types in the class we are extending. To check this, firstly substitute the object types for the type variables using `typeSubst`, then check equivalence using `equivTypes`.

Finally, we can consider the most general case which also allows *overriding*—giving a new value to a field specified in the superclass.

$$\frac{\begin{array}{c}\tau <: \tau' \qquad \Gamma \vdash e : \mathsf{Class}(\tau') \qquad (\tau_k'^{\mathsf{inh}}[\tau'/Y] = \tau_k^{\mathsf{inh}}[\tau/X])_{k \in 1..p} \\ (\Gamma, x : \tau \vdash e_i^{\mathsf{dec}} : \tau_i^{\mathsf{dec}}[\tau/X])_{i \in 1..m} \qquad (\Gamma, x : \tau \vdash e_j^{\mathsf{ovr}} : \tau_j^{\mathsf{ovr}}[\tau/X])_{j \in 1..n}\end{array}}{\Gamma \vdash \mathsf{class}(x : \tau)\ \mathsf{extends}(e : \mathsf{Class}(\tau'))\ \{\{\ell_i^{\mathsf{dec}} = e_i^{\mathsf{dec}}\}_{i \in 1..n}\}\ \mathsf{overrides}\ \{\{\ell_j^{\mathsf{ovr}} = e_j^{\mathsf{ovr}}\}_{j \in 1..m}\} : \mathsf{Class}(\tau)}$$

where

$$\tau = \mathsf{Object}(X)[\{\ell_i^{\mathsf{dec}} : \tau_i^{\mathsf{dec}}\}_{i \in 1..m} \cup \{\ell_j^{\mathsf{ovr}} : \tau_j^{\mathsf{ovr}}\}_{j \in 1..n} \cup \{\ell_k^{\mathsf{inh}} : \tau_k^{\mathsf{inh}}\}_{k \in 1..p}]$$
$$\tau' = \mathsf{Object}(Y)[\{\ell_j^{\mathsf{ovr}} : \tau'^{\mathsf{ovr}}_j\}_{j \in 1..n} \cup \{\ell_k^{\mathsf{inh}} : \tau'^{\mathsf{inh}}_k\}_{k \in 1..p}]$$

Here, we now distinguish a set of *overridden* labels which are common to both τ and τ', but appear in the overrides clause. Extending the previous rule to deal with this involves checking that each expression e_j^{ovr} for overridden labels ℓ_j^{ovr} has type τ_j^{ovr} in an environment extended with $x : \tau$.

Note that a plain class is actually a specialisation of the more general class construct, extending from the root class.

$$\mathsf{class}(x : \tau)\ \mathsf{extends}(\mathsf{root} : \mathsf{Class}(\mathsf{Object}(X)[])) \{\{\ell_i : \tau_i\}_{i \in 1..m}\}\ \mathsf{overrides}\ \{\}$$

You may want to implement it in this way to save yourself some coding. Alternatively, you are perfectly welcome to implement the rule for plain classes explicitly.

Exercise 5. *In `Source.scala`, implement the rules for classes by filling in the `RootClass`, `New`, and `Class` cases in the `typeCheck` function.*

[10 marks]

$$
\begin{array}{rcll}
[\![\mathsf{object}(x:\tau)\,\{\{\ell_i = e_i\}_{i\in 1..n}\}]\!] & = & [\{\ell_i = (x)\ \texttt{=>}\ [\![e_i]\!]\}_{i\in 1..n}] & \\
[\![\mathsf{root}]\!] & = & [new = (x)\ \texttt{=>}\ []] \qquad x \text{ is fresh} & \\
[\![\mathsf{fun}(x_1:\tau_1, x_2:\tau_2, \ldots, x_n:\tau_n)\,\{\,e\,\}]\!] & = & \lambda x_1.(\lambda x_2.(\cdots.(\lambda x_n.[\![e]\!]))\cdots) & \\
[\![e(e_1,\ldots,e_n)]\!] & = & [\![e]\!]([\![e_1]\!])\cdots([\![e_n]\!]) & \\
[\![\mathsf{new}\ e]\!] & = & [\![e]\!].new & \\
[\![e.\ell]\!] & = & [\![e]\!].\ell & \\
[\![e_1.\ell := e_2]\!] & = & \mathsf{update}\ [\![e_1]\!] :: \ell\ \mathsf{with}\ ((x)\ \texttt{=>}\ [\![e_2]\!]) \qquad x \text{ is fresh} & \\
[\![e_1.\ell := \mathsf{method}(x:A)\{\,e_2\,\}]\!] & = & \mathsf{update}\ [\![e_1]\!] :: \ell\ \mathsf{with}\ ((x)\ \texttt{=>}\ [\![e_2]\!]) & \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{let}\ x = [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!] & \\
[\![n \in \mathbb{N}]\!] & = & n & \\
[\![e_1 \oplus e_2]\!] & = & [\![e_1]\!] \oplus [\![e_2]\!] & \\
[\![!e_1]\!] & = & ![\![e_1]\!] & \\
[\![\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] & = & \mathsf{if}\ [\![e]\!]\ \mathsf{then}\ [\![e_1]\!]\ \mathsf{else}\ [\![e_2]\!] & \\
[\![\mathsf{true}]\!] & = & \mathsf{true} & \\
[\![\mathsf{false}]\!] & = & \mathsf{false} & \\
[\![x]\!] & = & x &
\end{array}
$$

Figure 6: Desugaring Terms and Objects

5 Desugaring to the Core Language

In the final part of this assignment, you will desugar Elephant into Core Elephant. As a result, you can evaluate Elephant programs using the evaluator you defined in Section 3, instead of writing another evaluator from scratch!

The *desugaring translation* is illustrated in Figures 6 and 7. These figures define desugaring as a function $[\![-]\!]$ that takes an Elephant expression and returns a Core Elephant expression. The exercises in this section ask you to implement this function in Scala.

In particular, you will implement the function

```
def desugar(e: Expr): Core.Expr
```

in `Desugar.scala`.

Desugaring functional constructs. We recommend starting with desugaring let-bindings, numbers, binary operators, negation, Boolean values, conditionals, and integers. Desugaring values is straightforward; for example, we can translate a number as follows:

```
case Num(n) => Core.NumV(n)
```

For cases such as if $-$ then $-$ else, it is necessary to call the desugaring function on each subterm, for example:

```
case IfThenElse(e1, e2, e3) =>
  Core.IfThenElse(desugar(e1), desugar(e2), desugar(e3))
```

Note that we have multi-argument functions and function application in Elephant, but only single-argument functions and function application in Core Elephant. However, multi-argument functions can be expressed as a sequence of λ-abstractions; consider again the *addTwo* function from Section 4:

$$\mathsf{fun}(x:\mathsf{Int}, y:\mathsf{Int})\,\{\,x+y\,\}$$

This can be expressed as two λ-abstractions:

$$\lambda x:\mathsf{Int}.(\lambda y:\mathsf{Int}.x+y)$$

$$
\begin{array}{l}
[\![\ \mathsf{class}(x:\tau)\ \mathsf{extends}(e:\tau')\ \{ \\
\qquad \{\ell_i^{\mathsf{dec}} = e_i^{\mathsf{dec}}\}_{i\in 1..m} \\
\}\ \mathsf{overrides}\ \{ \\
\qquad \{\ell_j^{\mathsf{ovr}} = e_j^{\mathsf{ovr}}\}_{j\in 1..n} \\
\}\ \ ]\!] \\
\end{array}
\quad = \quad
\begin{array}{l}
[\ \ new = (z)\ \texttt{=>}\ [\{\ell_i^{\mathsf{dec}} = (s)\ \texttt{=>}\ z.\ell_i(s)\}_{i\in 1..m} \cup \\
\qquad\qquad\qquad\qquad \{\ell_j^{\mathsf{ovr}} = (s)\ \texttt{=>}\ z.\ell_j(s)\}_{j\in 1..n} \cup \\
\qquad\qquad\qquad\qquad \{\ell_k^{\mathsf{inh}} = (s)\ \texttt{=>}\ z.\ell_k(s)\}_{k\in 1..p}] \\
\qquad \{\ell_i^{\mathsf{dec}} = (z)\ \texttt{=>}\ \lambda x.[\![e_i^{\mathsf{dec}}]\!]\}_{i\in 1..m} \cup \\
\qquad \{\ell_j^{\mathsf{ovr}} = (z)\ \texttt{=>}\ \lambda x.[\![e_j^{\mathsf{ovr}}]\!]\}_{j\in 1..n} \cup \\
\qquad \{\ell_k^{\mathsf{inh}} = (z)\ \texttt{=>}\ [\![e]\!].\ell_k^{\mathsf{inh}}\}_{k\in 1..p} \cup \\
\ ]
\end{array}
$$

where $\tau' = \mathsf{Class}(\mathsf{Object}(X)[\{\ell_j^{\mathsf{ovr}} : \tau_j^{\mathsf{ovr}}\}_{j\in 1..n} \cup \{\ell_k^{\mathsf{inh}} : \tau_k^{\mathsf{inh}}\}_{k\in 1..p}])$

s, z are fresh

Figure 7: Desugaring Classes

Similarly, we can write multi-argument function application as multiple single-argument applications. For example, $\mathit{addTwo}(10, 15)$ becomes $(\mathit{addTwo}(10))(15)$.

Desugaring object-oriented constructs. Since the object syntax for Core Elephant is actually more permissive, the mapping for the object-oriented constructs is rather direct. An object $\mathsf{object}(x)\ \{\{\ell_i = e_i\}_{i\in 1..n}\}$ is represented in the core language as $[\{\ell_i = (x)\ \texttt{=>}\ [\![e_i]\!]\}_{i\in 1..n}]$ – that is, the same *self* parameter x is used for all fields, and the field body is translated directly.

Field updates $e_1.\ell := e_2$ do not mention a *self* parameter, so we can use any variable as long as it is fresh. Thus, we generate a fresh *self* parameter x using `Utility.gensym`, and translate each of the subterms. The translation for methods is similar, but this time we use the *self* parameter specified in the method update.

Field access is translated directly, as objects have the same label sets in the source and core languages.

Exercise 6. *In `Desugar.scala`, implement the desugaring function for all constructs except `Class`, `Root`, and `New`.*

[10 marks]

Desugaring classes. Finally, we may desugar classes to the core calculus. You may notice that there is no separate construct for classes in the core calculus, but we may emulate them using plain objects.

The key technique we use to emulate classes is to endow objects with a distinguished method *new*, returning a new object which invokes the object's fields with the *self* parameter as an argument.

There are three constructs left to translate. The first is the root class, which translates to an object with a single field *new*, which returns the empty object. The second is $\mathsf{new}\ e$, which invokes the *new* method on the translated class e.

Finally, we can desugar classes themselves. A class has a distinguished method *new*, which has a fresh *self* parameter z. This method returns an object containing entries for each label ℓ_i each with self parameters s, invoking $z.\ell_i(s)$.

Suppose we have a class with self-type τ which extends a class $\mathsf{Class}(\tau')$. We would therefore expect τ to contain labels ℓ_j^{ovr} which are *overridden* by the class, and ℓ_k^{inh} which are *inherited* by the class.

We would also expect a set of labels $\{\ell_i^{\mathsf{dec}} : \tau_i^{\mathsf{dec}}\}_{i\in 1..m}$, which appear in τ and not τ'.

Each declared or overridden field is translated as a method taking a fresh *self* parameter, with the body of the method translated as a λ-abstraction taking x as its parameter, and with the body of the function being the translation of the body of the clause. Inherited clauses invoke the corresponding method of the translation of the superclass.

Finally, we can desugar "plain" classes as a special case of the more general class construct:

$$[\![\mathsf{class}(x : \tau)\ \{\{\ell_i = e_i\}_{i\in 1..n}\}]\!] =$$
$$[\![\mathsf{class}(x : \tau)\ \mathsf{extends}(\mathsf{root} : \mathsf{Class}(\mathsf{Object}(X)[]))\ \{\{\ell_i = e_i\}_{i\in 1..n}\}\ \mathsf{overrides}\ \{\}]\!]$$

Exercise 7. *In* `Desugar.scala`*, implement desugaring for classes by filling in the cases for the* `RootClass`*,* `New`*, and* `Class` *constructs.*

[10 marks]

A Rules for Subtyping

$$\boxed{\tau_1 <: \tau_2}$$

$$\frac{}{\tau <: \tau} \qquad \frac{(\tau_i' <: \tau_i)_{i\in 1..n} \qquad \tau <: \tau'}{((\tau_1, \ldots \tau_n)\ \texttt{->}\ \tau) <: ((\tau_1', \ldots, \tau_n')\ \texttt{->}\ \tau')}$$

$$\frac{\begin{array}{c}\tau = \mathsf{Object}(Y)[\{\ell_i : \tau_i\}_{i\in 1..(n+m)}] \\ \tau' = \mathsf{Object}(Y)[\{\ell_i : \tau_i'\}_{i\in 1..n}] \qquad (\tau_i[\tau/X] = \tau_i'[\tau'/Y])_{i\in 1..n}\end{array}}{A <: A'}$$

Figure 8: Rules for subtyping

Here, we include the rules we use for subtyping. Note that you don't need to use these, but we include them if you're interested, or wish to further understand what is going on.

B Changes

- V1.1: Restored the `swap` function to the starter code. Added missing type substitutions to the object and class typing rules.
- V1.2: Added missing substitution rules for if-then-else and function application in the core language.

  Added `Utility.scala` explanation, as well as explicitly stating that it contains `Gensym.gensym` from Assignment 2.

  Fixed a typo in the main text which incorrectly stated that "The root class always has type $\mathsf{Object}(X)[]$", which now reads "The root class always has type $\mathsf{Class}(\mathsf{Object}(X)[])$".

  Fixed commands to run sample solution.