Homework # 14
due Tuesday, June 10, 13:00

This is an optional homework assignment. If you do better on this assignment than on a previous assignment, this assignment will take the place of the latter. Please email a ZIP archive of your `src` directory to `malte.schwerhoff@inf.ethz.ch` before the deadline.

1 Reading

Read chapter 22 on *type reconstruction*, also known as *type inference*.

2 Overview

Your task is to implement type inference for a simple lambda in Scala[1]. The calculus corresponds to the basic lambda calculus extended with let-bindings and the Y-combinator (`fix`). The types are `Nat, Bool`, type variables and functions ($\rightarrow$ aka `Arrow`).

You are given a skeleton project for the implementation that defines the syntax and types of the lambda calculus, the basic classes and methods that you have to implement, and several unit tests illustrating the expected output of your type inference implementation. You should be able to follow the book quite closely; the only additions are the `fix` construct and transitive type variable substitution.

3 Skeleton Setup

3.1 Installation

- Download the skeleton project provided on the course website and unzip it into a directory. For the rest of this document, we assume that this directory is named `pancake`. The project uses Scala's de-facto standard build tool sbt[2] for dependency management, building and testing.
- Make sure that you have Java 1.7 installed.
- Download and install sbt. The latter essentially corresponds to putting the start script `sbt.bat`/codesbt.sh into the path.
- Open a command-line/terminal/shell in directory `pancake` and run `sbt`. You should see some output indicating that certain dependencies are being resolved and downloaded, including the required Scala version 2.10.3. After a while you should be able to use the sbt prompt, indicated by a leading `>` character.

[1] `http://scala-lang.org/`
[2] `http://www.scala-sbt.org/`

- Run `> project` (without the leading `>`) on the sbt prompt. The reply should be something like "`pancake-skeleton (in build file:...`".
- Run `> compile`. The project should be compiled, and you should eventually see something like "`[success] Total time:...`".
- Run `> test`. The output should include four groups called `TermTests`, `AbsTypingTests`, `DSLTests` and `TraverserTests`. All tests in groups `TermTests` and `AbsTypingTests` should be reported as ignored. At the end of the output it should say "`All tests passed`".
- Run `> run`. The output should be "`Welcome to homework 14`".
- It is recommended to use an IDE such as IntelliJ IDEA or Eclipse, although you can use a regular text editor in combination with sbt for development. Sbt can generate project configurations for both by either running `> gen-idea`, or respectively, `> eclipse`.
  If you want to use IntelliJ[3] you need to install the `Scala` plugin. If you want to use sbt's console from inside IntelliJ then you also need to install the `SBT` plugin (there might be two plugins, the right one is the one saying "Simple Build Tool" in its description). The course assistant uses IntelliJ together with sbt's console.
  If you want to use Eclipse, get the Scala-IDE[4].

3.2 Overview

You should only need to look at the directories `pancake/src/main/scala` and `pancake/src/test/scala`. The former contains the skeleton of the actual implementation, the latter a couple of unit tests. You should only need to change `main/scala/Types.scala`, which is where the type inference is to be implemented, and `test/scala/TermTests.scala`, which is where the main unit tests reside.

`Syntax.scala` contains the *abstract syntax tree* of our lambda terms, `Types.scala` contains the types and the main components of the type inference implementation, `PrettyPrinter.scala` contains a pretty printer for lambda terms, `DSL.scala` contains a few helpers that make it more convenient to create lambda terms, and `Pancake.scala` is essentially empty and can be used for experimenting.

Running `> run` on the sbt prompt executes the code in the body of `Pancake.scala`. Running `> test` runs the tests in the Scala files in `test/scala/`.

Any command in sbt can be prefixed with a tilde, e.g., `> ~test`. This tells sbt to invoke the command (here `test`) each time a file from the code base changes.

[3] `https://www.jetbrains.com/idea/`
[4] `http://scala-ide.org/`

4 Your Task (100pt)

Implement type inference by following chapter 22 of the book. Figures 22-1 and 22-2 are the most relevant parts, as well as sections 22.6 and 22.7, which deal with let-polymorphism. Your implementation should support all terms and types defined by the skeleton. All tests found in `TermTests.scala` should pass.

It should not be necessary to change the signature of classes and methods declared in the skeleton! It should also not be necessary to change `Syntax.scala`, `PrettyPrinter.scala` and `DSL.scala`.

1. Open `Types.scala` and look for places marked by `???`. This is where your code should go.
2. Open `TermTests.scala` to see which kind of output is expected from your implementation. The tests are ignored by default. To enable them, replace `ignore("..."` by `test("...")`. Initially, all tests will fail because of exceptions thrown by the unimplemented methods which you are supposed to implement.
3. Start implementing type inference by initially supporting only simple lambda terms. The test case titled `"very simple terms"` shows a few examples.

5 Bonus Task

This task is just for fun, but won't earn you additional points.

Can you modify your implementation such that it records and pretty-prints the types of variables bound in lambda abstractions as well? Here are some examples:

```
((lambda x : Nat . succ x) 0): Nat

(lambda f : Nat -> A . f 1): (Nat -> A) -> A

(lambda f : A -> B -> C .
  lambda x : A . lambda y : B .
    f x y): (A -> B -> C) -> A -> B -> C
```

`AbsTypingTests.scala` contains a few tests. In order to get similar output you will have to modify the pretty printer as well. Try solving this task by subclassing `TypeInferer` and `PrettyPrinter` in a way that allows you to reuse most of what has already been implemented.

6 Scala Primer

Scala is a hybrid language that mixes concepts from object-oriented and functional programming. It has a powerful, static type system and decent type inference. It compiles to

Java byte code, and thus, all Java/JVM libraries can be used. It supports closures/higher-order functions. It has a flexible syntax which makes it well-suited for embedded DSLs. It's standard library is documented[5] reasonably well.

Scala has Java-like classes (`class`), but the primary constructor is implicit. That is, the arguments to a class, e.g., `class Foo(s: String) println(s)`, are the arguments of the primary constructor, and the statements that occur inside the class (and not inside a method) are the body of the primary constructor. Arguments to the primary constructor are automatically stored in object fields.

Scala distinguishes between `var`s and *val*s. The former are mutable references which can be re-assigned to, whereas the latter are immutable references (similar to Java's `final` fields/variables). A declaration `var x: Int = 0` inside a class body declares a mutable object field, whereas `val x: Int = 0` declares an immutable one. The same declaration inside a method body declares a local var/val.

`case classes` are similar to Haskell's data types. Unlike regular classes, they can be constructed without `new`, and they can be deconstructed in pattern matching.

Scala encourages the use of immutable data structures. Hence, the default `List` and `Map` data structures are immutable. Mutable versions can be imported from `collections.mutable`. Scala has a very (very!) rich collection library and most collection classes support a variety of operations known from functional programming such as `map`, `filter` and `foldLeft`.

[5] `http://www.scala-lang.org/api/2.10.3/`