Software verification - Type Inference

Assignments for week 5

17^{th} May 2017

Goal

In this assignment you are going to implement the type inference algorithm $\mathcal{T}$ for the language *ML*. Programming such an algorithm from scratch takes quite some effort. That's why we provide some Java code that you can take as a starting point. For those who don't like Java, it is allowed to implement the algorithm in your favourite language. Below we describe the skeleton implementation and indicate what we expect you to add.

Package description

The predefined Java project consists of an number packages each representing an aspect of *ML*.

- The package `ast.expr` contains classes to represent the abstract syntax tree (AST) of *ML*. It consists of one base interface `Exp` that is implemented by all syntactical categories. For example, lambda expressions are represented by the class `Lam` that implements this interface. The interface `Exp` consists of a single methode `inferType`. All binary expression are represented by one and the same single class `BinExp` having the operator (of type `BinOp`) as an attribute. `BinOp` is an `enum` type with a constant for each binary operation. So called *constant-specific method implementations* are used to provide the types corresponding to those operations.
- The package `ast.type` consists of classes that represent types. Again, a base interface is introduced (`Type`) on which all concrete types are built. The `Type` interface contains several helper methods needed to implement a type inference algorithm. Moreover, `Type` extends the `Unify` interface containing methods that all together provide a unification procedure. A complicating factor with unification is that it takes two arguments (which both are of type `Type`) which have to be inspected both in order to decide whether unification succeeds or not. Usually, one uses pattern matching on both arguments to determine how to continue. In Java, we want to avoid pattern matching and choose *dynamic binding* as a case distinction mechanism. The way we implemented unification resembles the *Visitor Pattern*: the actual type categories of both involved

typed are determined by double polymorphic method invocation. Additionally, we decided to use java exceptions to indicate unification has failed.

- The `typing` package contains some elementary typing components, such as `Environ` to represent environments, `Subst` for substitutions , and `TVPool` to generate fresh type variables. As to the class `Environ`, we decided to allow for multiple declarations of the same variable. This is done to cope with scoping issues introduced by lambda expressions. Substitutions are represented as `Maps` (just like environments). To avoid implementing the composition of substitutions we effectively maintain one global substitution the can be used/changed at any time. This explains, for example, why type substitutions for the variable case are implemented differently compared to the standard algorithm as presented during the lectures. In short, we no longer apply the resulting substitution just after typing a subexpression, but defer this operation until we have to perform a unification step. The class `TVPool` in the `typing` package provides a simple mechanism to generate new (fresh) type variables.
- Finally, there are two helper packages `examples` end `main` which contain concrete examples that can be used for testing.

Assignment

The implementation of the typing algorithm is not yet complete. Your task is to add the following:

- An implementation of `inferType` for **if** expressions.
- A representation for both tuple expression (including the selectors **fst** and **snd**) and tuple types. This includes that you provide an implementation of all abstract methods.
- A representation of **let** expressions including a definition of `InferType`. At first instance, you can restrict yourself to the monomorphic case.
- An implementation of *lists*. Hint: Use your solution for part 2 of last week's assignment.

Optional challenge

Implement a polymorphic variant of **let** expressions.

Handing in

Hand in all `.java` files via Blackboard. **Deadline Monday** 29^{th} **May, 12:30**.