Computer Science 320 (Fall, 2009)
Concepts of Programming Languages

Assignment 7: Evaluation Strategies and Type Inference

Out: Tuesday, November 17, 2009
Due: Monday, November 30, 2009

In this assignment, you will complete the implementation of two interpreters and a type inference algorithm for Mini-Haskell. You will make changes to the `Eval` and `Ty` modules, and will submit an additional file, `tests5.mhs`.

This assignment assumes you have completed the `Unify` module in **Assignment 5** and the `Eval` and `Ty` modules in **Assignment 6**. You may use either the official solutions provided for these assignments, or your own solutions (or any combination of the two). In the latter case, you will need to copy the skeleton code for the `ty` function from the `Ty` module provided for this problem set.

You must retrieve the new versions of `Parser`, `Exp`, and `Val` supplied for this problem set (you may also want to download `tests3.mhs` and `tests4.mhs` to test your code). These are identical to the versions used in the previous assignment, with the exception of new code and definitions necessary for handling lambda abstractions:

```
data Exp = ...
         | Lam [String] Exp
           ...

data Val = ...
         | VLam [String] Exp (Env Val)
           ...
```

You are not allowed to modify any of the existing data type definitions unless a problem states otherwise or explicitly directs you to change a definition.

Note: All function definitions at top level must have explicit type annotations. Unless otherwise specified, your solutions may utilize functions from the standard prelude as well as material from the lecture notes and textbook. You may **not** use any other materials without explicit citation.

Problem 1. (15 pts)

In this problem, you will implement an evaluation algorithm for Mini-Haskell that uses the call-by-value (CBV) environment model as defined in Chapter 13 of the lecture notes. You will do so by modifying the existing definitions in the `Eval` module.

(a) In the `Eval` module, modify the definition of the `appVals::Val -> Val -> Error Val` function so that it can handle the case in which the first argument is an evaluated lambda abstraction, and the second is an argument for that lambda abstraction. You only need to handle lambda abstractions with a single variable (that is, cases where the list of strings under the `Lam` constructor has exactly one element). You may extend your code to handle lambda abstractions with tuples for extra credit as part of your solution to **Problem #4** below.

(b) In the `Eval` module, modify the definition of the `ev::Env Val -> Exp -> Error Val` function so that it can evaluate lambda abstractions.

You have now implemented a complete call-by-value interpreter for Mini-Haskell, and should be able to test it using `parseEval` from the `Main` module. The program in `tests4.mhs` computes a list of prime numbers using bounded recursion.

Problem 2. (60 pts)

In this problem, you will implement a complete interpreter for Mini-Haskell that uses the call-by-name (CBN) *substitution* model, as defined in Chapter 13 of the lecture notes.

As usual, you only need to handle `let` bindings and lambda abstractions with a single variable. You may extend your code to handle lambda abstractions and `let` bindings with tuples for extra credit in **Problem #5** below.

(a) In the `Eval` module, import the `Unify` module and add an `instance` declaration that makes `Exp` a member of the `Substitutable` class. You may assume that all variable names in a program are unique (for extra credit, you may change your code so that it works even without this assumption in **Problem #6**), but you must not substitute a bound variable *under* its own corresponding lambda or `let` binding. Think carefully about the checks that need to be performed when `subst` encounters a lambda abstraction or `let` expression. There are two ways you can approach this problem:

- assume that substitutions of type `Subst Exp` have exactly one element and maintain this property in any substitutions of type `Subst Exp` you construct (this is the easier option);
- add a function `remove::String -> Subst a -> Subst a` or `remove::[String] -> Subst a -> Subst a` to the `Unify` module that removes the elements in the substitution that correspond to the variable name(s) in the argument(s).

(b) In the `Eval` module, implement a function `appValExp::Val -> Exp -> Error Val` that evaluates a value applied to an expression. You should not evaluate the second arguments of the short-circuited boolean binary operators (i.e. `((&&) False)` and `((||) True)`). Note that for some cases, `appValExp` will need to build substitutions and call `subst`. **When you need to perform an evaluation, call `ev0`, not `ev`.** You will complete the implementation of `ev0` in part (c) below. For convenience, you may call `appVals` in some cases, but do this very carefully. You should not evaluate any subexpression that does not need to be evaluated according to the evaluation rules of the call-by-name substitution model.

(c) In the `Eval` module, modify the body of the `ev0::Exp -> Error Val` function so that it evaluates *all possible* expressions (as usual, excepting lambda abstractions and `let` bindings with multiple arguments) according to the call-by-name evaluation *substitution* model, as defined in Chapter 13 of the lecture notes (Hint: For `let` bindings, it may be easier to first convert each one into an equivalent application of a lambda abstraction to an argument, and then to use `appValExp`, or even a recursive call to `ev0`).

(d) Create a file named `tests5.mhs`, and inside it write a small program on which the call-by-value interpreter defined by `ev` diverges (runs forever), but on which the call-by-name interpreter `ev0` terminates.

Problem 3. (25 pts)

In this problem, you will complete a few parts of the full type inference algorithm for Mini-Haskell. You will complete the already partially-implemented function `ty::Env Ty -> FreshVars -> Exp -> Error (Ty, Subst Ty, FreshVars)`, found in the `Ty` module available on the course website.

The `ty` function binds fresh *type* variables to any new variables it encounters in lambda abstractions or let-bindings. Each time the function processes some node in an expression tree, it may need to unify the types of the children of that node, and so, it accumulates these substitutions, returning them to the caller. The cases for variables, `let` bindings, and applications have already been completed for you, and you may use them as a guide in completing the remaining parts of the algorithm.

(a) Add the base cases for expressions (the empty tuple, the empty list, integers, boolean values, and operators) to the definition of the function `ty`. You may simply return the fresh list of variables unchanged, and an empty substitution.

(b) The case for `if` expressions in the definition of `ty` is not complete. The function

```
tys::Env Ty -> FreshVars -> [Exp] -> Error ([Ty], Subst, FreshVars)
```

is used to obtain the types for all three subexpressions simultaneously. Perform the remaining checks and unifications in order to complete this part of the definition. Remember that any substitutions you might generate in the process (including the one that has already been generated when type checking the three subexpressions) need to be considered in every subsequent unification.

(c) The case for tuples in the definition of `ty` is not complete. Complete this case.

(d) The case for general lambda abstractions in the definition of `ty` is not complete. Complete this part of the definition. Because a new variable is encountered, you will need to obtain a fresh *type* variable, and bind this expression variable to the fresh type variable in the environment. The subexpression should be type-checked under this new environment.

Remember that the argument type of the abstraction is the same as the type of the variable over which the abstraction is defined. You may want to look at the part of the definition of `ty` for `let` bindings for guidance.

Problem 4. (*10 extra credit pts)

Extend your solution to **Problem #1** so that it handles lambda abstractions with multiple arguments.

Problem 5. (*20 extra credit pts)

Extend your solution to **Problem #2** so that it handles lambda abstractions and `let` bindings with multiple arguments.

(a) Modify `subst` in the appropriate manner.

(b) Modify `appValExp` in the appropriate manner. Be certain that you do not evaluate the argument passed to a lambda abstraction that takes unit (an empty tuple) as an argument.

(c) Modify `ev0` in the appropriate manner.

Problem 6. (*20 extra credit pts)

You will implement a transformation on expressions that ensures that the implementation of `subst` in **Problem #2** works correctly for all programs.

(a) Implement a recursive function `unique::[String] -> Env String -> Exp -> (Exp, [String])` that takes an expression and gives a unique name to every variable in that expression, ensuring that no two lambda abstractions or let bindings are over a variable with the same name. You will need to maintain a list of fresh variable names. This list must also be returned with the result because an expression may have multiple branches, and variables cannot repeat even across branches. The environment is used to associate old variable names with new ones. More points can be earned if your function handles `let` bindings and lambda abstractions with multiple arguments.

(b) Modify `evalExp::Exp -> Error Val` so that an expression is evaluated only after being transformed using `unique`.