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

Assignment 6: Basic Evaluation and Type Checking

Out: Thursday, November 5, 2009
Due: Monday, November 16, 2009

In this assignment, you will implement an interpreter and type checker for a subset of Mini-Haskell, which is itself a subset of of Haskell. You will make changes only to the `Eval` and `Ty` modules.

The code provided to you requires a parser library, *Parsec*. When using Hugs, you will probably need to supply the `-98` command-line directive in order to run the code. You can check whether you are able to run the skeleton code simply by loading the `Main` module and attempting to run `parseEval "tests0.mhs"`. This should evaluate to a message indicating that the evaluation function is not yet implemented.

Note that 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. There are, however, crude `show` functions for expressions, and you may want to use them at times when debugging (think about returning a subexpressions as a string in an error message).

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. (30 pts)

In this problem, you will implement an interpreter for expressions without variables or `let` bindings. Because the parsed code is not being type checked before being interpreted, errors may occur during evaluation, so you will need to use `Error Val` as the return type for the various evaluation functions. Familiarize yourselves with the `Exp` data structure in the `Exp` module (`Exp.hs`) that represents the abstract syntax of the language. The `Val` data structure found in the `Val` module (`Val.hs`) represents evaluated expressions.

(a) In the `Eval` module, implement the body of the function `appOp::Oper -> Val -> Error Val` that takes an operator and a value, and when the operator is defined on that value, returns the result.

If the operator is binary, take advantage of the `Partial::Oper -> Val -> Val` constructor, which can represent a partially applied built-in function. For example, `appOp` might return a value like `Partial Plus (N 4)` if it must apply the binary operator `Plus` to the single value `N 4`.

If the operator is not defined on that value, the function should return an error. Be careful with list operators such as `Head`, as they are not defined on empty lists. When in doubt, you can use the real Haskell interpreter to determine how this function should behave.

(b) In the `Eval` module, implement the body of the `appBinOp::Oper -> Val -> Val -> Error Val` function, which takes an operator and two values, and returns the resulting value if it is defined. Otherwise, it returns an error.

You may assume that `Equal` is defined only on integers. You are not required to implement equality on boolean, list, and tuple values, but you may do so for extra credit as part of your solution to **Problem #4** below.

(c) In the `Eval` module, implement the body of the `appVals::Val -> Val -> Error Val` function, which takes a pair of values where the first value is either a unary operator, a binary operator, or a partially applied binary operator. You should not need to use any functions other than those you defined in parts (a) and (b).

(d) In the `Eval` module, implement the body of the `ev0::Exp -> Error Val` function. This function should handle all expressions that do not contain `let` bindings or variables. Thus, it should evaluate all base cases (such as operators, booleans, integers, tuples, and the empty list) as well as `if` statements and applications. For all other cases, `ev0` may return an error. In evaluating tuples, you may find it convenient to use the `mapError::(a -> Error b) -> [a] -> Error [b]` function, available in the `Err` module.

Note: You *should not* evaluate both branches of an `if` statement, and this is tested in the file `tests1.mhs`.

You should now be able to test the interpreter on some input. When using Hugs, it should be sufficient to load the `Main` module, and to run `parseEval "tests0.mhs"` and `parseEval "tests1.mhs"`. You may, of course, write and try evaluating your own test programs.

Problem 2. (30 pts)

In the module `Env`, you will find an implementation of an environment data structure which can be used to store the values (and types) of variables. You will use this data structure to represent evaluation and type checking contexts. You will now implement an interpreter that can handle variables and `let` bindings.

(a) Implement the body of the `ev::Env Val -> Exp -> Error Val` function, which can handle expressions that contain `let` bindings and variables. You only need to handle let bindings with a single variable (that is, cases where the list of strings under the Let constructor has exactly one element). You may extend your code to handle let bindings with tuples for extra credit as part of your solution to **Problem #4** below.

The base cases should be similar to those of ev0, except for the case of a variable. A variable evaluates to the value with which it is associated in the environment. If it is not found in the environment, it is not bound, and an error should be returned. When encountering lambda abstractions, remember to store the current environment inside them.

(b) Modify the `evalExp::Exp -> Error Val` function in the `Eval` module to call `ev` instead of `ev0`.

You should now be able to test your interpreter using `tests2.mhs`.

Problem 3. (40 pts)

You will now implement a type checker for a subset of Mini-Haskell. Familiarize yourself with the abstract syntax for types in the `Ty` module.

(a) Implement a function `tyOp::Oper -> Ty` which returns the type of an operator. You may assume that `(==)` can only be applied to integers, and that only integer lists can be constructed (which means, for example, that the type of `[]` should indicate that it is an integer list).

(b) Implement a function `ty0::Exp -> Error Ty` that can successfully type check all primitives (operators, integers, and booleans), tuples, `if` statements, and applications.

Since there are no type variables, you may use derived equality `(==)` on types. Remember that both branches of an `if` statement must have the same type, and that an `if` condition must have a boolean type. For application, you may need to pattern match on the type of the function in order to check that its type matches the type of its argument.

Your solution should be able to type check `tests0.mhs` successfully, and should reject `tests1.mhs`, as it is not well-typed.

(c) Implement the body of the `ty1::Env Ty -> Exp -> Error Ty` function. This function can handle expressions that contain `let` bindings and variables. You only need to handle `let` bindings with a single variable (that is, cases where the list of strings under the `Let` constructor has exactly one element). You may extend your code to handle `let` bindings with tuples for extra credit as part of your solution to **Problem #4** below.

(d) Modify the `typeCheck::Exp -> Error Ty` function in the `Ty` module to call `ty1` instead of `ty0`.

Your type checker should now be able to handle `tests2.mhs`.

Problem 4. (*40 extra credit pts)

(a) In the `Eval` module, modify the definition of `appBinOp::Oper -> Val -> Val -> Error Val` so that equality is also defined on boolean, integer, tuple, and list values. Two tuples are equivalent only if they are of the same length (including the case where both are of length zero); two lists are equivalent only if they are of the same length and each of the corresponding values in the two lists are equivalent.

(b) In the `Eval` module, modify the definition of `ev::Env Val -> Exp -> Error Val` so that it can handle `let` bindings with tuples. You should first evaluate the expression that must be bound to the variables. If the `let` binding is for a tuple, this expression must evaluate to a tuple of the same length. Otherwise, `ev` should return an error. Once the value tuple is obtained, it is only necessary to take the current environment and bind each element of the tuple to the corresponding variables. You may want to use the `updEnvL::[(String, a)] -> Env a -> Env a` function, found in the `Env` module, to accomplish this. Then, the body of the `let` binding can be evaluated under the new environment.

(c) In the `Ty` module, modify the definition of `ty1::Env Ty -> Exp -> Error Ty` so that when the equality operator is applied to booleans, integers, tuples, or list values, type checking succeeds as long as both values are of the same type (you may ignore the case in which the operator is applied partially to only a single value).

(d) In the `Ty` module, modify the definition of `ty1::Env Ty -> Exp -> Error Ty` so that lists of any type can be constructed and expressions will still type check (you will need to handle the list operators in a special way, as you did with equality in part (c) above).

(e) In the `Ty` module, modify the definition of `ty1::Env Ty -> Exp -> Error Ty` so that it can handle `let` bindings with tuples.

A complete type inference algorithm for Mini-Haskell that can handle variables, `let` bindings, lambda abstractions, and polymorphic functions and values (such as `[]` and `(==)`) can only be defined with the help of unification. To solve the following parts, you will need to import the `Unify` module from the previous assignment into the `Ty` module. These functions will be used on the next assignment to define a full type-checking algorithm.

(f) Write an `instance` declaration in the `Ty` module that makes `Ty` a member of the `Substitutable` type class.

(g) Write an `instance` declaration in the `Ty` module that makes `Ty` a member of the `Unifiable` type class.

(h) Notice the type definition `type FreshVars = [Ty]`. Define a value `freshTyVars::FreshVars`, an infinite list of type variables in which no type variable is ever repeated.